SAP AG

Differential Buffer for a Relational Column Store In-Memory Database

Project for University Carlos III of Madrid

Autor
Jorge Gonzalez Lopez
Research in the SAP HANA Database Team
Dietmar-Hopp-Allee 16
69190, Walldorf, Germany
jorge.gonzalez.lopez@sap.com

Tutor
Jose Daniel Garcia Sanchez
Computer Architecture and Technology Area
Avda Universidad Carlos III, 22
28270, Colmenarejo, Madrid, Spain
josedaniel.garcia@uc3m.es

Contents

Figures

1. Abstract

At the present the financial and the analytical reporting has taken importance over the operational reporting. The main difference is that operational reporting focus on day-to-day operations and requires data on the detail of transactions, while the financial and analytical reporting focus on long term operations and uses multiple transactions. That situation, added to the hardware evolution, have determined the relational databases over the time. One of the different approaches is the actual SAP HANA database. This database focus on the financial and the analytical reporting without the use of the Data Warehouses. But it also provides good capabilities for the operational reporting. That was achieve through the use of a column store structure in main memory. But in order to prepare the data in the database, it holds up the insertion performance. This document studies the possibility to use a buffer in a prototype based in the SAP HANA database architecture, with the goal of improve that performance. In order to compare the impact in the system of the addition of a buffer, multiple approaches has been implemented, tested and carefully compared to each other and also to the original prototype.

2. Introduction

Nowadays, in the modern business generates a huge quantity of data. Sometimes the set of data is so large and complex that the storage, analysis, search, and other operations… become too difficult. The trend, as it can be appreciated from the beginning of the databases history, is that the size of the data increases with the time. For example, on 2012 the Square Kilometer Array (SKA) of radio telescopes will produce over one Exabyte (2^{60} bytes) every day. How to handle all that data, has become a crucial point for business.

Initially, the classical systems were intended to handle thousands of users and transactions with very high insertion load and very selective point queries, in order to produce operational reporting. But it is only in the long term, using very long sets of data when a company can understand its own business, through the use of what is called financial and analytic reporting. As the financial and analytic reporting started to increase in importance, and since its complexity growths with the size of the data, the system structure had to change. That introduced the Data Warehouses, a new part of the system in charge to process special workloads. The problem was that the data in the Data Warehouses was an outdated processed version of the data from other parts of the system, hence the data was replicated. A renovation of the paradigm was something necessary, because there was room enough to optimization. This renovation of the system came because of two events.

One was the semiconductor memory becomes cheaper and chip densities increase quickly, that allowed the possibility to develop large databases in memory. This type of databases was very attractive since the data accessed in main memory can provide a much better response time and transaction throughput than the conventional databases.

The other one was that some researchers started to go in deep in other ways to store the data. That propitiated the apparition of the column store, a new conception to store the records of a relational database. Instead of group the values by entries, it does it by attributes. That provide a totally improved performance regarding to the analytical and financial performance. But in exchange it introduced an overhead in the insertion process.

A combination of both events, and the development by the Hasso Plattner Institute and the company SAP, lead to the SAP HANA database. A relational column store in-memory database. This database was intended to replace the combination of the old systems and the Data warehouses with a unique system capable of both types of workloads.

But the problem of the overhead in the insertion was still present in the system. It was believed there was the possibility of optimize the system through the addition of a buffer in the SAP HANA database. In order to improve the insertion performance, and try to avoid as much as possible the impact in the other queries performance, multiple buffer strategies are going to be studied and tested in this document.

In order to achieve the goals mentioned above, in the following sections, the document will do a thorough study and explanation of the multiple aspects related to the project. First of all, it will establish the background to understand the concepts explained later on, concepts such as the Online Transaction Processing, the Online Analytic Processing, Row store, Column store,

etc… Then it will show the disadvantages of the old system, how they adapted to the evolving situation. This will establish the reasons to create SAP HANA database, its advantages respect analytic and financial performance, and its problems with the insertion performance. The next chapter is focus on how to recreate the situation, without having to use the real product. Once that is done, it will show the different ways to interact with the system, through the use of different operations. And after, the creation of the buffer, with multiple strategies in order to try to optimize those operations. The succeeding chapter emphasizes in the way the prototype and the operations were implemented. Specially shows the communication between the different parts of the system to execute the operations. And after that chapter, comes the evaluation. The first that is explained there are the different scenarios. The way the tests were created, and the different parameters that can be configured. And then, a detailed analysis of all the different test with all the different approaches and combinations is done. And last, the conclusion of all the evaluation process, a combination of solutions and how the system could be improved in a future.

3. Related work and evolution to the current situation

In this section we will go from the first beginnings of the *Relational Database Management Systems* (RDBMS) to one of the latest and most revolutionary RDBMS, SAP HANA. In order to establish a solid background to the evolution of these systems, some concepts like OLTP and OLAP scenarios, row storage and column storage, etc… will be explained during the section. The main goal is to provide to the ability to the reader to understand the key characteristics of SAP HANA.

3.1. Beginning of relational database management systems

In order to see the evolution of the RDBMS we have to go back to the 1970s. At that moment, the dominant database management system was the navigational model, also known as CODASYL [16]. The CODASYL used fixed length registers and a number of fixed attributes. The registers are connected to each other through static links, pointers. All this simplified the implementation, but the model was very inflexible, if you wanted to run an operation that was not expected and implemented before, you had to recompile the whole schema again.

In contraposition to the CODSYL model, Edgar Codd, an employee of IBM in California outlined a totally new approach in "A relational Model of Data for Large Shared Data Banks" [1]. The new model was based in the relation between the data, forgetting about how the data structure. The idea behind was to allow the user to only focus in what he wanted to know and not in how that knowledge was storage.

The fundamental structure of the model was the *relation,* also known as *table*. The table was form by two dimensions, one for the rows (tuples) and the other for the columns (attributes). Each tuple is considered a set in the mathematical sense, an unsorted collection of different elements. In order to difference a tuple from another, Edgar Codd establishes the concept of *key.* A key is an attribute (or a set of attributes) that unique for each tuple, and so it allows to difference the tuples [17].

The new model was not support at the beginning for IBM, who instead chose IMS, since they invested a big amount of effort and money in it [17]. But a group in the university of Berkley (California) ,leaded by Michael Stonebraker, believed in the new model and so started to create a new system, the *Ingres* [18]. This work and the efforts of other large database vendors motivated IBM to develop their own database system, the *System R* [19] [20], which supported multiuser and a structured language, *SEQUEL* [19] [20], which with the time would end called *SQL*. The first version of the product was released in 1974.

3.2.Row store

The companies started to use the database systems in order to storage all the information. To achieve that, they developed different applications that fitted their requirements. Some examples are: a payroll application that could handle the salaries of the employees, an order entry application that would allow registering the sale of a seat for a performance, a bank application for handle the money of the clients of a bank, etc… The processing of one of these items is a business transaction [2]. At that moment the priority was to handle as many transactions as possible.

In order to achieve that goal each time a transaction was completed and ready to be saved the RDBMS pushed all its attributes continuously. This strategy is commonly known as *row store*, because in the logical visualization of the two dimensional table, each tuple was a row. The process of insertion in a row store is shown in the figure 1. To insert a new tuple, the space where it will be locate has to be found, then all the values of the new tuple are pushed in that place.

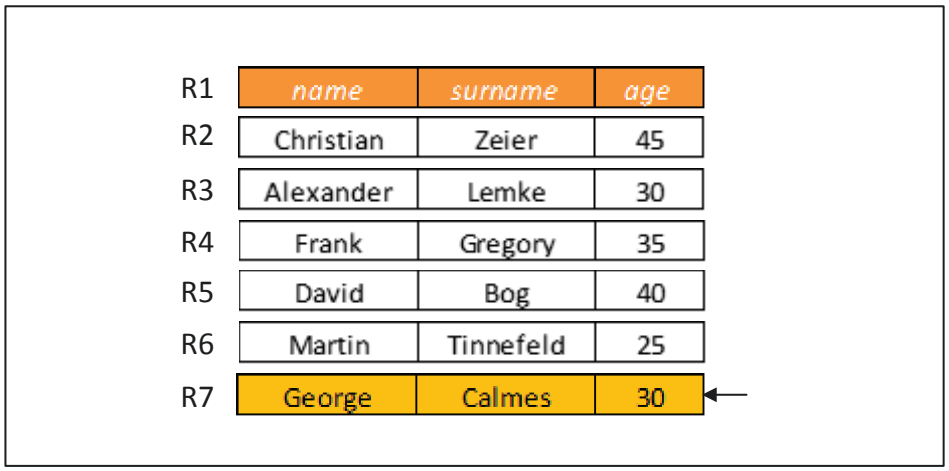


Figure 1: Insertion of a tuple in row store

The insertion of data is basic in order to have storage all the information needed to run or even improve our business. But the logical though is that if we save that information, is because we are going to need it later. All this interaction with the RDBMS is done through asking it for information or ordering it to save information. This is done through the use of what is known as *queries*. Because of the relational model, normally the queries can be classified in two different groups.

The first group is known as *Online Transaction Processing* (OLTP). This is because is transaction oriented; typically we classified here the insertion of data or the retrieval of transactions [21]. For example, if we have a table that are the different employees of a company a typical OLTP query would be to add a new employee or consult how much is an specific employee earning. The access pattern for this type of workload is shown in the figure 1. As we can appreciate the row store is ideal for OLTP because for each tuple we access, we get multiples attributes.

The second group is called *Online Analytical Processing* (OLAP). This is because normally are generated by financial or analytical applications. OLAP queries are largely attribute-focused

rather than transactional-focused [21], but only involving a small number of columns of the table. For example, following the previous type of table, if we have a table that stores all the employees of a company and we want to know how much we are spending in paying the salaries of all of them, which would be classified as an OLTP workload. The figure 2 shows the access pattern for this type of queries. The problem is that since the system is oriented to process a tuple-at-a-time we have to access to all the employees, discard all the attributes that are not needed, and then access to the next. The problem remains in the amount of I/O traffic spend in attributes that are not going to be used. This makes OLAP queries to be slower than OLTP, and the locks of the analytical queries might result to increase the running time of OLTP queries.

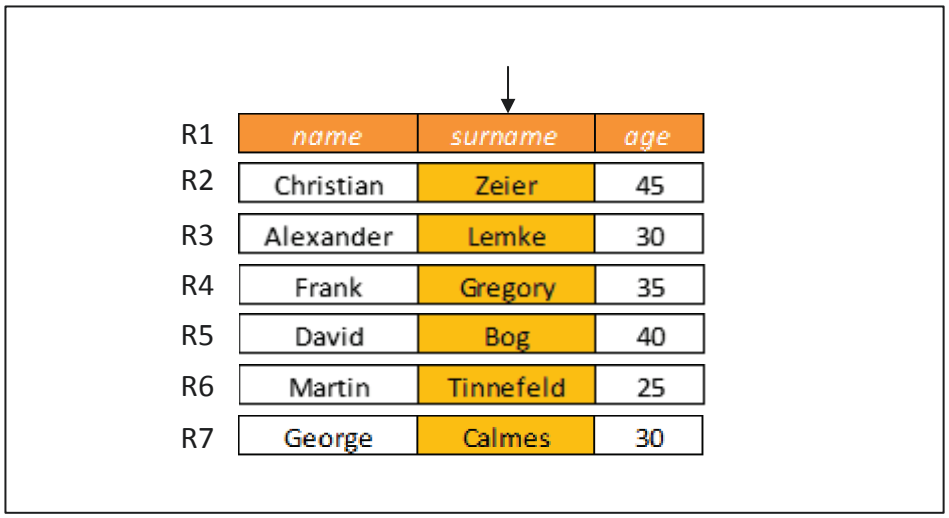


Figure 2: Read of attributes in row store

3.3.Data Warehouses

Given this new situation, appears the distinction between *operational* reporting and *informational* reporting [24]. Operational reporting requires a more detailed, up-to-date data, and read or updates a few records which are accessed typically on their primary key. Operational reporting workloads are usually related to OLTP and because of the reasons we saw in the previous section they were transferred to the traditional databases that started to be called Operational Databases. And informational reporting usually needs historical, summarized and consolidated data in order to make long-term decisions. The informational reporting workloads are query intensive with mostly ad hoc, complex queries that can access millions of records. This type of workload is related to OLAP and is transferred to the data warehouses [23]. The relation between those systems is shown in the figure 3.

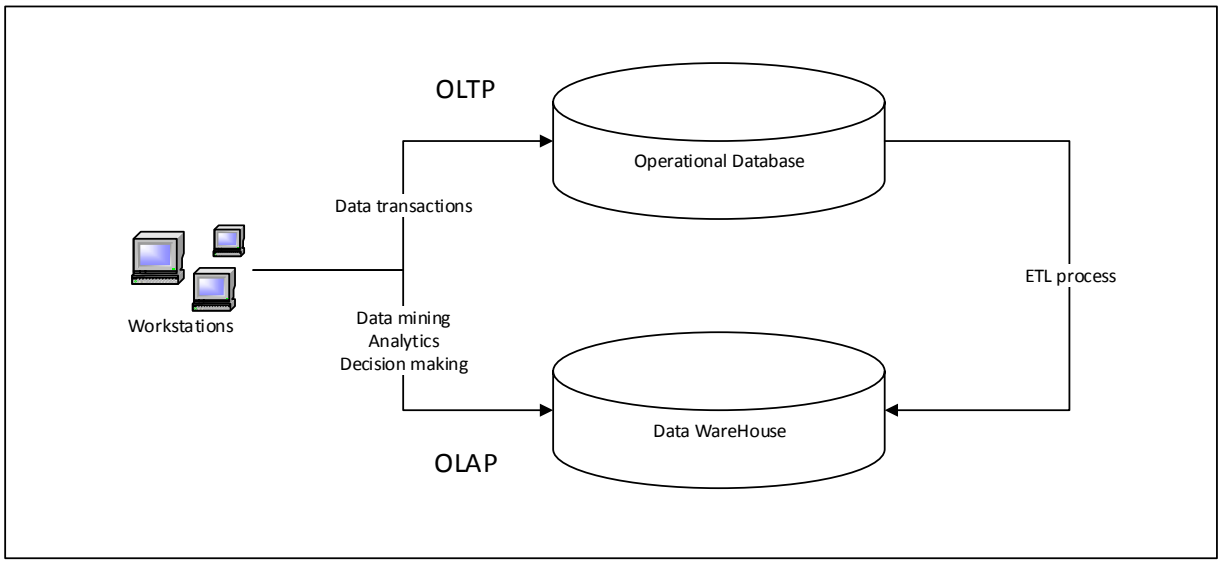


Figure 3: Relation between Operational Databases and Data Warehouses

A data warehouse is a "subject-oriented, integrated, time-varying, non-volatile collection of data that is used primarily in organizational decision making" [22]. In the middle of 90's data warehousing have been used in many industries: manufacturing (for order shipment and customer support), retail (for user profiling and inventory management), financial services (for claims analysis, risk analysis, fraud detection…), etc… [23].

In order to understand how they achieved those goals, and to understand its later tradeoffs, the explanation of the communication process between the operational databases and the data warehouses is mandatory. This process is called ETL process (Extract, transform and load). This process consists first in the *extraction* of the data from external sources. This extraction can come from different sources or from a single one. After comes the *transformation* of the data. Since usually the data comes from multiple sources, there is a high probability of errors (inconsistent field lengths, inconsistent descriptions, missing entries and violation of integrity constrains). Once the data is prepared, comes the *load* into the data warehouse. In this phase additional processing may be done, for example, sorting, summarization, aggregation, indices, etc… All these operations prepare the data for its future workloads. Once all those calculations are done, the data is properly inserted in the data warehouse, usually through batch loads techniques. The ETL process is explained more in detail in [23].

The use of data warehouses provided a solution for poor performance of the informational reporting in the traditional systems but at the same time it produced some difficulties. First of all, we have to design before-hand the granularity of the operational data, with might not be desirable. And the most important issue, the updates. Since all the data is replicated to the data warehouse, we have to decide if we want to run updates between the operational database and the data warehouse more frequently and reduce the query performance drastically; or in the other hand run the updates less frequently in order to do not affect the performance, but that would make our historical data out-dated which might not be an option.

3.4.Column Store

Although the operational reporting was an essential part of business, they started to realize that it was only through informational reporting that they were able to understand their own business. Using data warehouses implied some important tradeoff in that area. For example, if we are a large company that sell a specific product, probably we are going to design the system for calculate the total sales for, as maximum, each month. That means there is no way we can know which, from the four weeks of a month, was the less productive. Another example would be that we could not be sure how the sales are going until the month actually ends. At first, those situations do not look very important, but are an essential part of understand how your business works.

Informational reporting was more related to the OLAP workloads, and at the same time, OLAP workloads were more attribute-focused than transactional-focused. Knowing that, an old strategy proposed in 1985 was started to be taken into consideration as a possible solution to the situation exposed above, the *column store* [26]. It proposes to store all the attributes of each column continuously, instead of store the attributes of the same tuple continuously as the row store does. This process is shown in the figure 4. Mainly, you have to split a tuple into the different attributes, and insert each one of them in the correspondent column.

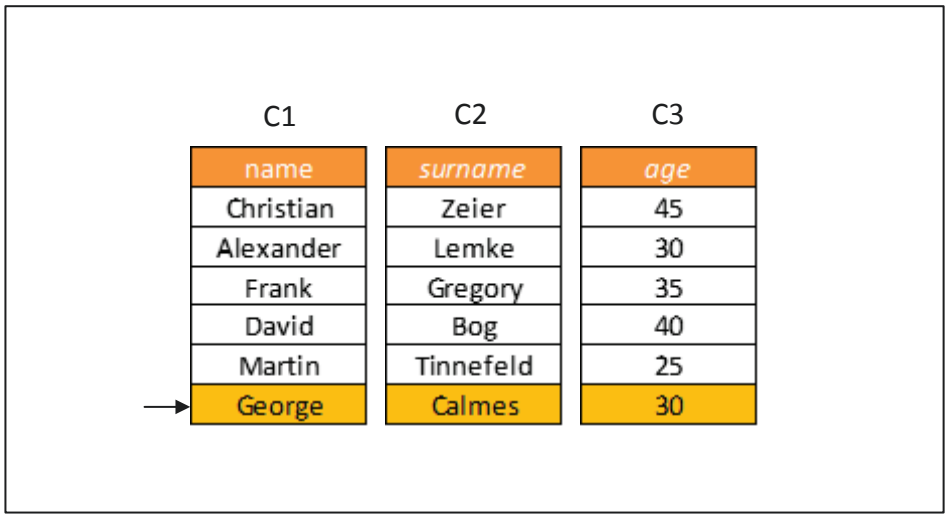


Figure 4: Typical access pattern in column store for OLTP

Regarding to the *Online Transaction Processing* (OLTP) workloads performance in column store there are a few aspects that need to be distinguished. Normally in the OLTP workloads, as explained before, a few tuples are involved and those usually are searched by their primary key. Also the entry and modification of data is considered in this type of workload. Considering this, the access pattern for those workloads is shown in the figure 4. It is less effective than in the row store because when you want to access all the attributes of the same tuple, in order to rebuild it, you have to do multiples reads for access to the multiple attributes.

Although the column store seems to be poorly optimized for this type of workloads, there is an aspect that has to be taken into consideration. According to the Moore's law [3], the number of transistors which can be inexpensively placed on a chip continue to double every two years or so. Although the clock speed improvement has ceased years ago, the Moore's law is still valid in

a different aspect, the increase of cores in a single CPU. That is a very important aspect for the column store, because that type of architecture is very scalable regarding to the number of cores. The easiest example is to dedicate a single core for each column when OLTP has to be done. So, even OLTP workloads do not perform as good as in row store, the difference can be reduced.

But the most important aspect, where the column store definitely outstands, is in the OLAP workloads. As explained before, OLAP queries are largely attribute-focused [21]. The figure 5 shows the access pattern for this type of workloads. The great advantage of column store is that for OLAP workloads, it can avoid bringing into memory unneeded attributes. This could mean, if the system is properly design, that would allow us to run informational reporting out from the data warehouses. Without the needed of have duplicated data, or preconfigure the workloads beforehand. So for example, if we would like to know how the sales of a company went the last week, there is no need for wait for the ETL process to take place and then consult the data warehouse.

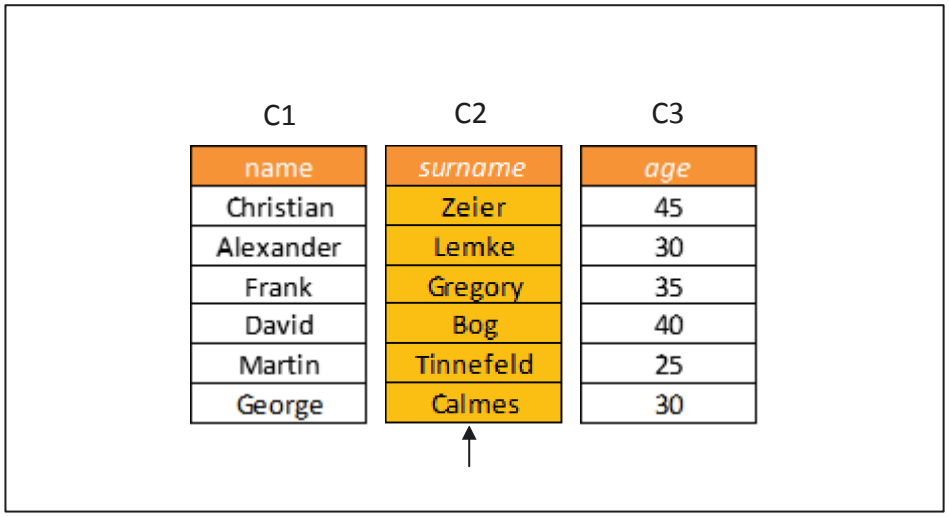


Figure 5: Typical access in column store for OLAP

But this situations creates a conflict between optimize a system for OLTP workloads, and focus on the operational reports, or optimize for OLAP workloads, and focus on the informational reports. For many years a solution was to have separate systems that would be optimize for specific tasks.

3.5.Hardware Evolution

Something to take in consideration for how the systems could work now days is the latest hardware evolution.

Some years ago the main memory was so expensive it was only used as a kind of cache for where the rest of information was stored on the hard disk. This was because the only way to store all the information from a data base was using hard disk, in concrete it was necessary multiple hard disk to accumulate enough space, because the price per MB in hard disk was much more cheaper than in main memory.

The evolution of hardware affected the main memory and the hard disk sizes, so at the moment it is not difficult to find hard disks of terabytes and main memory slots of 16GB. This made possible to consider doing with main memory what was done with hard disk many years ago, and build clusters of main memory in order to create systems with even a terabyte of main memory. Although this solution is more expensive than the use of traditional storage, it could be used in some scenarios such as real time applications where transactions must be completed instantaneously, because the access time to main memory is orders of magnitude less than for hard disk.

Considering the exposed above we should think about if it is not time to change the roles of hard disk and main memory for the sake of performance. Use the main memory as main storage and only use the hard disks as backup system (because of the non-volatile properties).

Having this concepts in mind, Peter Boncz et al. [5] create an in-memory database system. Based in the concepts of how spatial locality affects to the read performance in main memory of the modern systems [6]. This is considered one of the first in-memory relational databases; its results and functionality were motivation to the following in-memory relational databases that were developed later.

3.6.Hana

All the previous research and evolution explained before was taken into account by SAP. Around 2005, SAP had three separated products related to data bases. TREX (Text Retrieval and Extraction): it was a search engine that ended adding in-memory column store. P*Time: a light-weight OLTP RDBMS based in row store. And MaxDB: a relational database that provided persistence to TREX and P*Time. At that moment, SAP had different specialized products, each one optimized for different purpose. But the question was, it that was still necessary.

Anja Bog et al. [12] points out that for many real scenarios some workloads cannot entirely fit in the classification of OLTP or OLAP, sometimes it is not obvious when an application domain belongs to one or another. Krueger et al. [11] even expose some mixed workloads that are generated by a single application. For example the demand planning, the available to promise or the dunning applications do not fit exactly in one type of workload, as we can appreciate in the following diagram [11].

	Demand Planning	Sales Order Processing	Available to Promise	Dunning	Sales Analysis
Granularity of Data	Transactional	Transactional	Transactional	Transactional	Pre-Aggregated
Operations on Data	Read & Write	Read & Write	Read & Write	Read & Write	Read-Only
Preprocessing of Data	No	No	No	No	Yes
Timeframe of Data	Historical & Recent	Recent Only	Historical & Recent	Historical & Recent	Historical & Recent
Update Cycles of Data	Always Up-to-Date	Always Up-to-Date	Always Up-to-Date	Always Up-to-Date	Cyclic Updates
Amount of Data per Query	Large	Small	Large	Large	Large
Complexity of Queries	High	Standard	High	High	High
Predictability of Queries	Medium	High	Medium	Medium	Low
Response Time of Queries	Seconds	Seconds	Seconds	Seconds to Hours	Seconds to Hours

OLTP Characteristics are colored light grey OLAP Characteristics are colored dark grey

Figure 6: OLTP and OLAP characteristics of different applications

Taking the mixed workloads in consideration, SAP started to consider the idea of taking away the different specialized system and combine them all in one unique system capable of run operational reporting as informational reporting. This means that it would be optimizing for OLTP as for OLAP as well.

Having that goal in mind, SAP started to develop a hybrid system. A previous work [4] showed that having a hybrid system, a specialized part for OLTP workloads and another part for OLAP workloads, could perform considerably well for both scenarios. This new product is called HANA, and is where the rest of our work is going to be related to. The figure 7, which was taken from [9], shows an overview of the architecture of HANA which is going to be explained in the following sections.

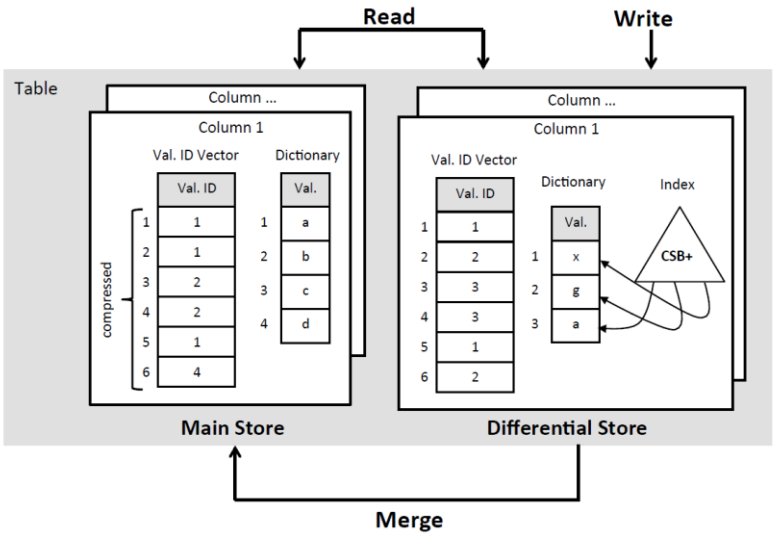


Figure 7: SAP HANA architecture of data representation [9]

3.6.1. Architecture

HANA is an in-memory RDBMS organized in columns. Each table is divided in columns, and each column has part of the information store in the main store and the other part in the differential store. Since the information is not replicated, the read operations happen simultaneously in both parts of the systems. Also the read operations only happen in the differential store, which at some point commits all those values into the main store through the merge operation. How these operations perform and their relations with the different parts of the system are going to be explain in the following sections. Although a more specific explanation can be found in [9], [15], [28] and [29].

3.6.1.1. Main Store

The main store is the largest part of the system, since it is going to store all the historical data. This part of the system is only going two types of operations; large amount of read operations and occasionally the addition of the new values coming from the merge process. Since the predominant operation is going to be the reads, this part of the system is mainly focused in be read-optimized, particularly for OLAP workloads.

One of the factors that have more repercussion in the performance in this section of the architecture is the use of data compression. Compression is use for three reasons: 1) Reduce the space of use in main memory (or in disk), it allows to keep more data in a single node, this way the traffic between the CPU and the memory is reduced [7], and in consequence increase the speed of query processing [14]. 2) Other reason is if we make aware to the query engine the type of compression is being use it can process data directly without computing-intensive decompression. 3) Data compression techniques exploit data redundancy, this makes column store a better scenario for compression than the row store, because all the data within a column has the same type and normally, since it is the same attribute of many tuples, has low information entropy.

The main trade-off of using compression is that the more we compress the data, the bigger will be the cost to de-compress the data. But nowadays the CPU speed grows at a rate of 60 percent each year, while the access time to memory increases less than a 10 percent per year. This trends defense the usage of data compression.

The main store uses different techniques for data compression. The main compression is done by using dictionary encoding. Dictionary encoding replaces each value with an index that is the position that takes the value in the dictionary. The figure 8 shows an example, the months of the year could be store in a dictionary and be replace with the positions in the dictionary (in this case we inserted them in order but is not necessary).

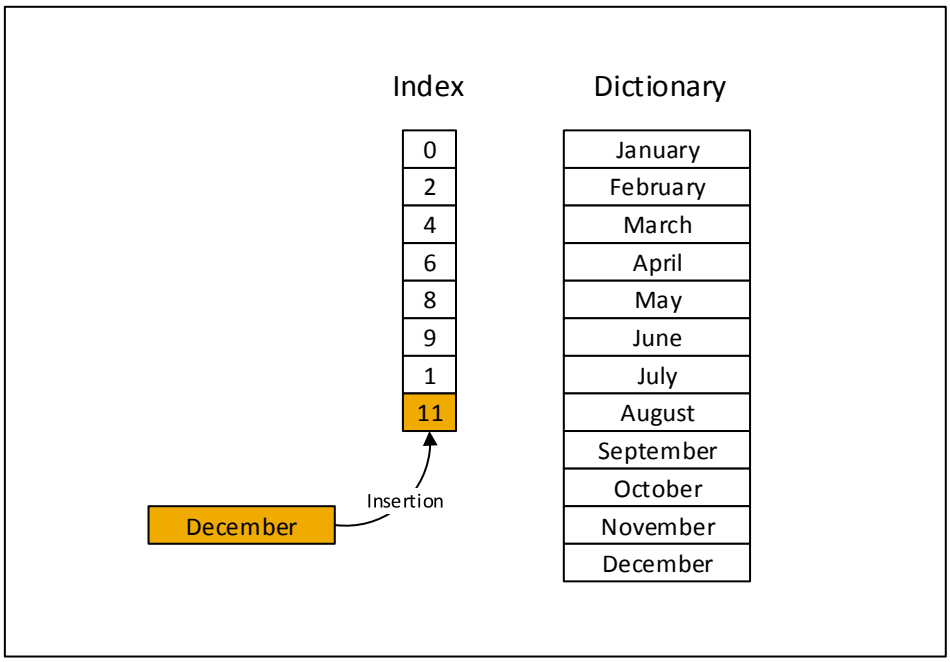


Figure 8: Insertion process in a dictionary compressed column

Even though the use of dictionary compression is useful in most of the cases, sometimes it affect in a negative way to the performance. For example, when the dictionary does not fit in the CPU cache or when the size of the un-coded column is smaller than the size of the index and the dictionary together [13]. This special case is taken into consideration and if the compression is not necessary is not used.

But when the dictionary compression is used correctly, it eliminates the redundant storage of values within a column and only redundancy of the index value referencing to the same value happens. This second redundancy is normally eliminated by using run-length encoding. Run-length encoding (RLE) transforms a sequence of repeated values in the following pair <value, number of occurrences>. In the figure 9, we can appreciate an example using the previous situation. We have stored the index as a pair of values, so in the first position we have three “January”, then only one “March, after two “May”, and so on… Since the dictionary coded the months, we can be sure there are never going to be more than twelve different values, and using RLE the size of the index is never going to be bigger than twelve even if we insert hundreds of values. Using RLE we eliminate the redundancy within the index and compress further the size of the column.

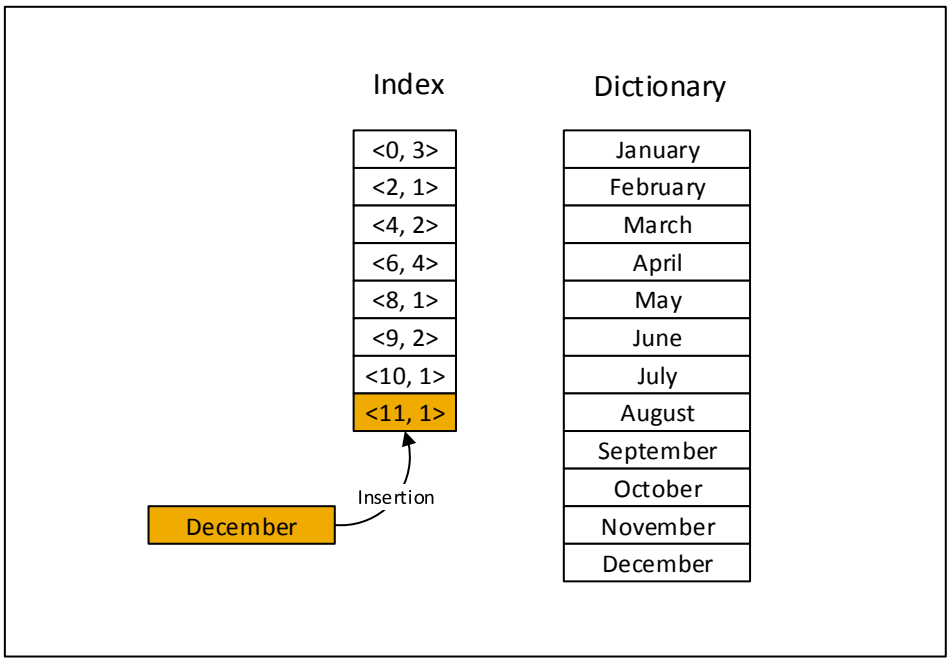


Figure 9: Insertion process in a dictionary compressed column with RLE

Sometimes, depending on the relation of the data, sparse coding or cluster coding are considered instead of RLE encoding [27].

The last optimization to take in consideration is that the dictionary is sorted. This allows us to run binary search over the dictionary, which specially improves the performance over large dictionaries.

So, given the state of the main store explained before, the look up for a value process would be the following. First, we perform a binary search over the dictionary. Once we find it, the position in the dictionary will be the value we have to search in the index. Then, we look up for that value in the index. If the index has RLE compression, then we will get as many values as were inserted.

As we can appreciate this process is much faster than if we would have every value uncompressed continuously stored. First because of the compression, that allows us to compare more values for each memory read. And second, because it is always faster to compare numbers than characters. If we have to choose between compare a number of 5 digits or a word of 5 characters, the number would take only one comparison while the word would take 5 comparisons, one for each character.

Regarding to the write process, although for reasons that are going to be shown later the conventional insertion of data does not take place here, the hypothetic process for insert one value can be found in the figure 10. Keeping the previous example, if we want to insert the last month of the year, December, in the real main store, where the dictionary is sorted, the method would be the following. First we would need to find the position where we should insert the value in the dictionary. Once we find it, we have to update in the index all the values that are below the value that we inserted in the dictionary. In our example, we would need to find in the index all the values from 2 (February) to 10 (September) and update them to the new value, that

would be exactly adding one, meaning that February now should be 3, January should be 4, and so on…

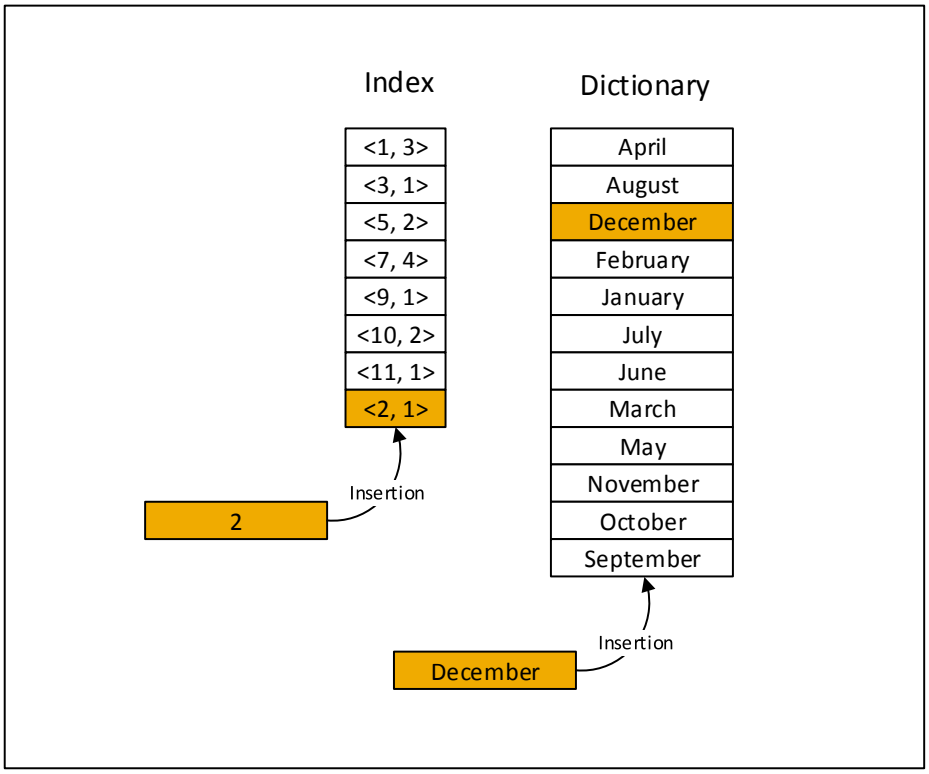


Figure 10: Main Store

We can appreciate there is a problem here. Because if we have *n* values in the dictionary and we insert a value that belongs to the first position in the dictionary, we would need to update every value in the index. That is the reason of creating a different part that would receive the write operations, and the main store would only get new data through the merge process. The specifics of that process can be found in [9] and [15].

3.6.1.2. Differential Store

Because the main store would not perform well enough for insertions, SAP came up with a hybrid solution, adding a different part to the system that would be solving the problem of the main store. This is the reason why there is not direct insertion of data to the main store, because all of them are done first in the differential store.

A more specific view of the differential store can be found in the figure 11. This part of the system is very similar to the main store. It is also organized in columns, so it is still oriented to support large amounts of query workloads. The data is compressed for the same reasons are explained above: for save space, reduce traffic between CPU and memory, speed up queries without the need of uncompressing the data, etc… The biggest modification between the main store and the differential store is how it manages the dictionary. Now, instead of having the dictionary sorted allowing to do binary search but with its tradeoffs in the insertion, the dictionary it is not sorted and it uses a CSB+ tree [8] for index the values of the dictionary.

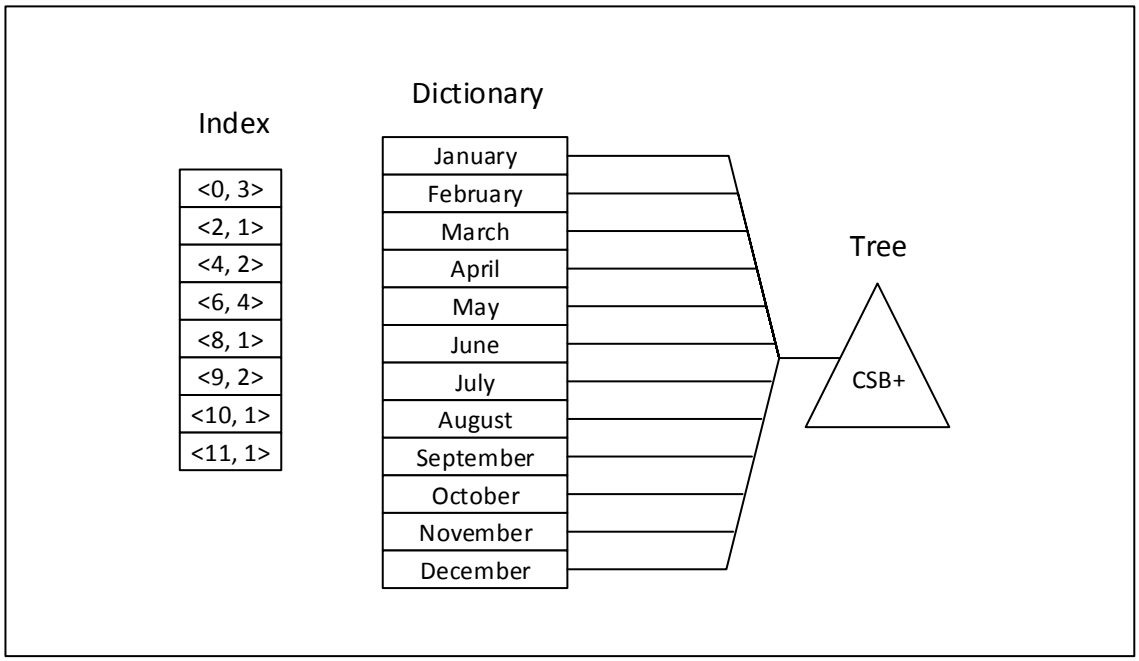


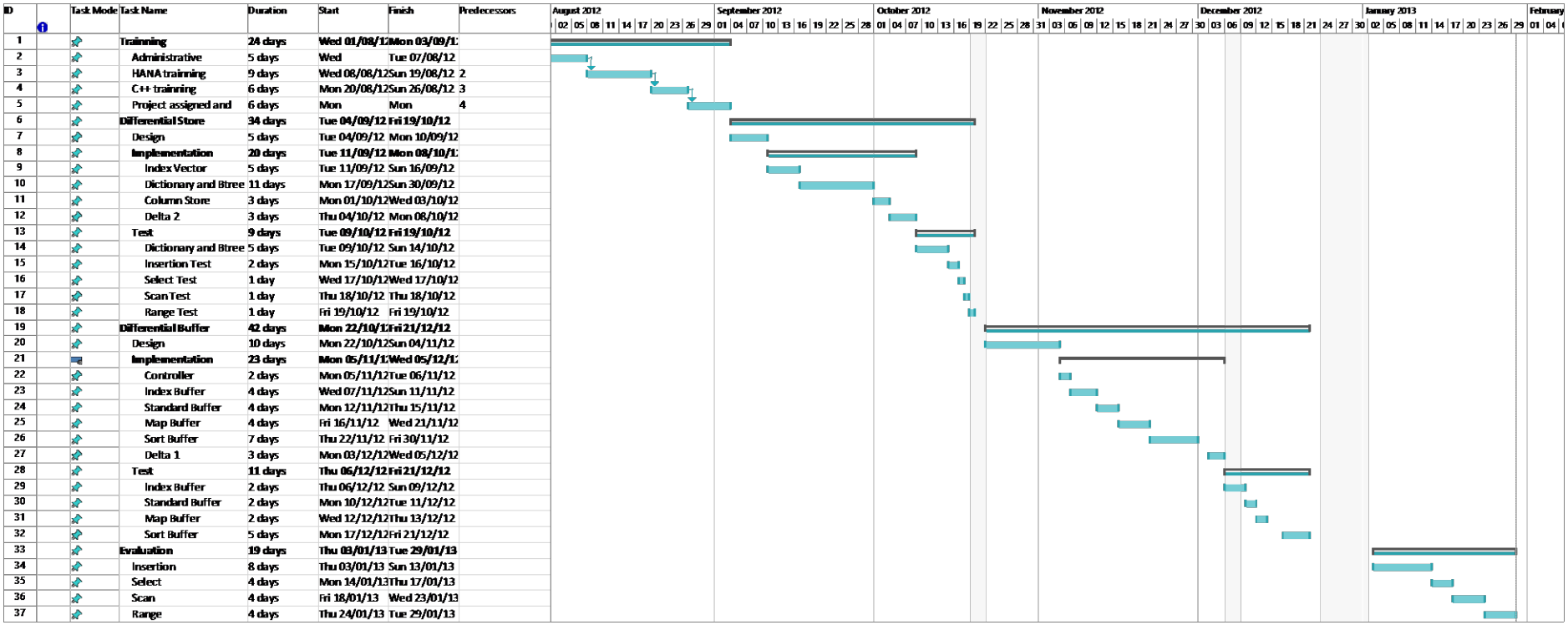
Figure 11: Differential Store

Given the previous structure for the differential store, the look up for a value process would be the following. First, we would need to find in the tree structure the value we are looking for, the complexity of this is $\log_b$ of the number of elements, for *b* the order of the tree. Once we find the position of the value we just need to look it up in the index for find how many times it was inserted. This gives a look up performance almost as good as in the main store.

But the big difference comes for the insertion of data in the differential store. Here, since the dictionary is not sorted, instead is indexed. When we want to insert a value we only have to add the value at the end of the dictionary, and then update the tree with the new value. After that is done, we only have to add it to the index. Here, in contrast to the main store, when we add new values, the only structure that needs to be updated is the tree, which will need to split nodes, balance, etc… And this is faster, because the previous order of the values in the dictionary remains and hence the index does not need to be updated.

4. Planification of the project

The following Gantt chart shows the scheduled times for the diferent stages in the project. It shows the progress from August to the end of January, that it was the lapse of time where the project took place.



ID	Task Mode	Task Name	Duration	Start	Finish	Predecessors
1		**Trainning**	**24 days**	**Wed 01/08/12**	**Mon 03/09/12**	
2		Administrative	5 days	Wed	Tue 07/08/12	
3		HANA trainning	9 days	Wed 08/08/12	Sun 19/08/12	2
4		C++ trainning	6 days	Mon 20/08/12	Sun 26/08/12	3
5		Project assigned and	6 days	Mon	Mon	4
6		**Differential Store**	**34 days**	**Tue 04/09/12**	**Fri 19/10/12**	
7		Design	5 days	Tue 04/09/12	Mon 10/09/12	
8		**Implenentation**	**20 days**	**Tue 11/09/12**	**Mon 08/10/12**	
9		Index Vector	5 days	Tue 11/09/12	Sun 16/09/12	
10		Dictionary and Btree	11 days	Mon 17/09/12	Sun 30/09/12	
11		Column Store	3 days	Mon 01/10/12	Wed 03/10/12	
12		Delta 2	3 days	Thu 04/10/12	Mon 08/10/12	
13		Test	9 days	Tue 09/10/12	Fri 19/10/12	
14		Dictionary and Btree	5 days	Tue 09/10/12	Sun 14/10/12	
15		Insertion Test	2 days	Mon 15/10/12	Tue 16/10/12	
16		Select Test	1 day	Wed 17/10/12	Wed 17/10/12	
17		Scan Test	1 day	Thu 18/10/12	Thu 18/10/12	
18		Range Test	1 day	Fri 19/10/12	Fri 19/10/12	
19		**Differential Buffer**	**42 days**	**Mon 22/10/12**	**Fri 21/12/12**	
20		Design	10 days	Mon 22/10/12	Sun 04/11/12	
21		**Implenentation**	**23 days**	**Mon 05/11/12**	**Wed 05/12/12**	
22		Controller	2 days	Mon 05/11/12	Tue 06/11/12	
23		Index Buffer	4 days	Wed 07/11/12	Sun 11/11/12	
24		Standard Buffer	4 days	Mon 12/11/12	Thu 15/11/12	
25		Map Buffer	4 days	Fri 16/11/12	Wed 21/11/12	
26		Sort Buffer	7 days	Thu 22/11/12	Fri 30/11/12	
27		Delta 1	3 days	Mon 03/12/12	Wed 05/12/12	
28		Test	11 days	Thu 06/12/12	Fri 21/12/12	
29		Index Buffer	2 days	Thu 06/12/12	Sun 09/12/12	
30		Standard Buffer	2 days	Mon 10/12/12	Tue 11/12/12	
31		Map Buffer	2 days	Wed 12/12/12	Thu 13/12/12	
32		Sort Buffer	5 days	Mon 17/12/12	Fri 21/12/12	
33		**Evaluation**	**19 days**	**Thu 03/01/13**	**Tue 29/01/13**	
34		Insertion	8 days	Thu 03/01/13	Sun 13/01/13	
35		Select	4 days	Mon 14/01/13	Thu 17/01/13	
36		Scan	4 days	Fri 18/01/13	Wed 23/01/13	
37		Range	4 days	Thu 24/01/13	Tue 29/01/13	

5. Design of the Prototype

In order to find a solution to the insertion performance, and by the way improving the general performance for the OLTP workloads, in this paper we propose the use of a buffer which is going to be called a differential buffer or delta buffer.

This section is the first section which will explain in an abstract level the prototype. Once the motivation of the development of the buffer is establish, the different use cases of the prototype will be exposed. Then the general architecture is explained, showing the functionality of the different parts of the prototype. After, the differential store and the differential buffer are going to be explained in detail.

5.1. Motivation

In the previous chapter the general functionality of HANA was exposed. The use of an additional part, the differential store, in order to speed up the insertions but without having any considerable repercussion in the query performance. This was achieved through the combination of an unsorted dictionary and the addition of an index structure for the dictionary.

Although the insertion is faster in the differential store than in the main store, this performance decreases with the time. This is because the bigger the differential store is, the slower the insertion process gets. The main reason is because of the dictionary which is indexed by the tree, even though the dictionary does not need to modify for each insertion, the tree does. And when the tree is considerably big, the process to find the right place in the tree, insert the new value and keep the tree balanced is very slow. This situation is what motivated the research and realization of a buffer that is going to be explained in the following sections.

5.2. Use Cases

The use cases are the basic operations that are required to achieve a goal. In this case, there are four basic operations and through the use of those, more complicated operations can be achieved. The four operations that allow the interaction with the system are:

- Insert: This operation allows to the prototype to add incoming data. There are two different situations for the inserts, first of all if the buffer is not activated the data is divided in the different attributes, then each attribute is compressed and inserted. But if the buffer is activated, the data is divided in the different attributes and is inserted directly into the buffer. Based in the example of the figure 12, an insertion would be to introduce one person new, with all its attributes.

  Once the buffer the buffer reaches its maximum size, the merge operation is triggered. The merge is the process that happens when we empty the buffer and commit all the values to the differential store.

- Select: The select operation allows to lookup for all the tuples that have a specific value in a specific attribute. It returns in a row format all the tuples that match the conditions. So for example, based in the scenario that shows the figure 12, the select operations can select all the people that are 30 years old. The idea of this operation it is to be very selective, normally because a specific tuple is being looking for.

- Scan: The scan operation allows selecting all the attributes of a specific column. It will return in a row format all the attributes of the selected columns. So for example, based in the situation that shows the figure 12, the scan operation can return all the people on the table but selecting only the needed information. This operation avoids returning unneeded attributes in the results.

- Range: The range operation allows selecting all the rows that have an attribute within a specific range of values. It will return all the tuples that match the condition in a row format. For example, using the scenario exposed in the figure 12, the range operation can find all the people that are in a range of age from 30 to 35 years old, both included.

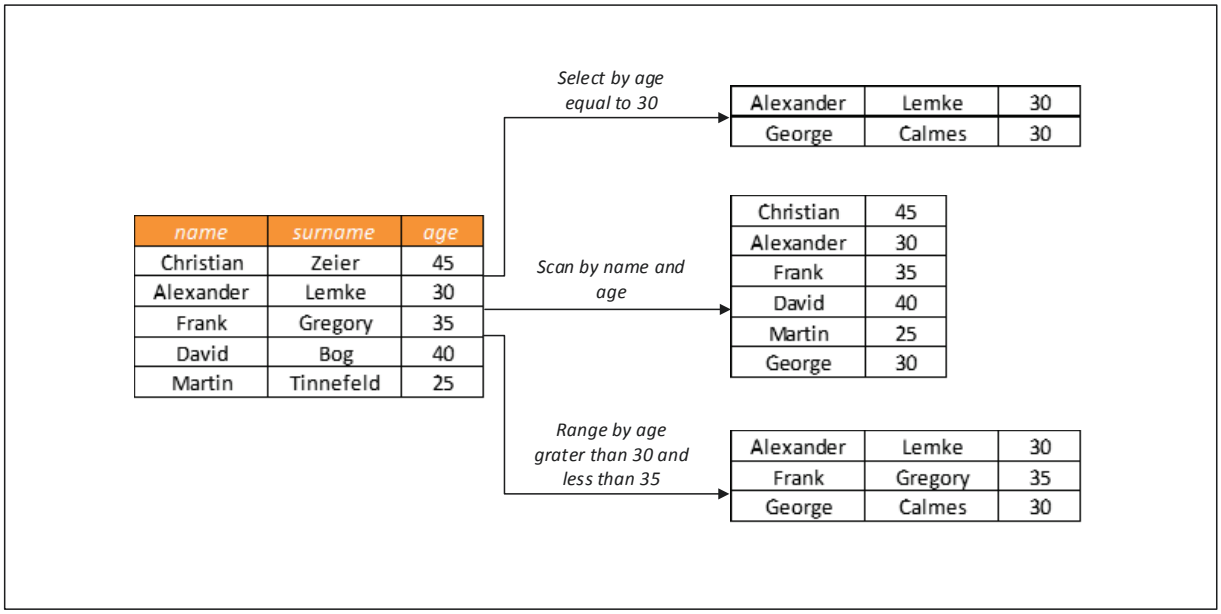


Figure 12: Examples of use cases

The insert operation is the only one implemented from all the operations that allow modifying the state of the system. This is because the other operations, delete/update, normally are implemented using flags for each attribute. So for example, in case the system wants to delete an attribute it sets the *invalid* flag, and if the system wants to update a value first sets the *invalid* flag and then inserts the new value. All the values set as invalid are discard in the merge operation. Our prototype does not implement that operation because the idea it was to keep it simple. Also to see the improvement in the insert operations there is no need to check the delete/update.

Regarding to the rest of operations, the choice of use those three is because most of the more complex operations are a set of combinations of those three. For example, using the scenario of figure 12, to retrieve only the age of Martin a combination of scan and select can be used.

Also, all the operations return the values in a row store format. This is because for data representation this format suits better than the column store. And because normally all the relational databases systems retrieve the tuples in row format, because is easier to handle it in the application level.

5.3.Prototype Architecture

In this section, a general view of the architecture used in our prototype is going to be shown. The idea behind the buffer is that all the incoming data is going to be store there, instead of in the differential store. The differential buffer is going to preserve the column store strategy that was proposed in the other parts of the system. But in contrast it is going to be optimized for fast inserts and deletes.

Since the goal is to improve the insertion operations, and based in the HANA architecture exposed in the previous chapter, it can be appreciated that the only part that could affect to the tests is the differential store. This is because, regarding to the previous architecture, the differential store was where the insertions were taken place.

The figure 13, based in the overview of the architecture of HANA [9] exposed in the previous chapter, shows the architecture and the main functionality of the prototype in a very abstract level. The prototype consists in two parts: the differential store which functionality is the same that was told in the HANA architecture, and the differential buffer. The differential buffer is the part where that processes the insertion operations and is going to store the data without compression, since this data is going to fill more space in memory the size of the buffer is going to be remarkable smaller than the differential store, but in the other hand is expected to improve the insertion times because the unneeded of process the data for compression. Also since we cannot be sure when the read operation are going to occur, there is no way to know if there is data in the buffer. This means the look up for a value have to take place in both parts in the system.

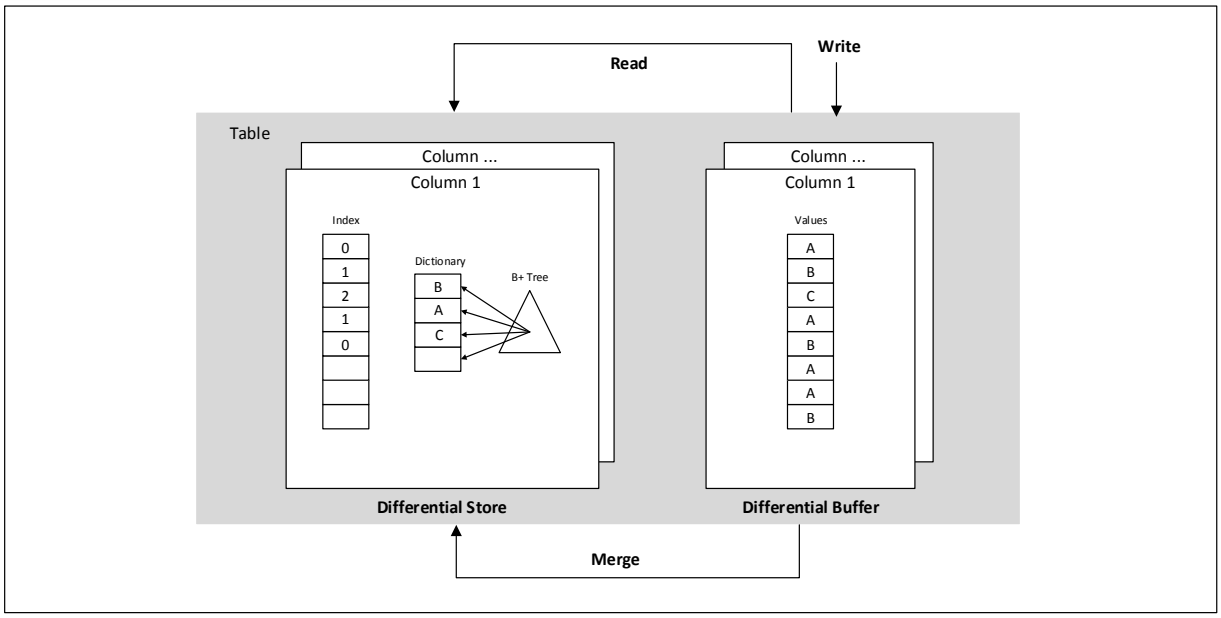


Figure 13: Architecture of prototype

Although the basic functionality of the prototype is explained above, a more specific view of the prototype can be found in the figure 14. The figure shows the two different parts of the system, but also the main components of each part. The prototype will always have a differential store, but the use of the buffer is optional. The motivation of this is because the performance of the prototype when uses the buffer and when does not it is what is going to be in compare. Regarding to the differential store, it is a set of column stores and each column store controls the communication between the index, the dictionary and the btree, corresponding to one column. And the differential buffer can implement five different strategies.

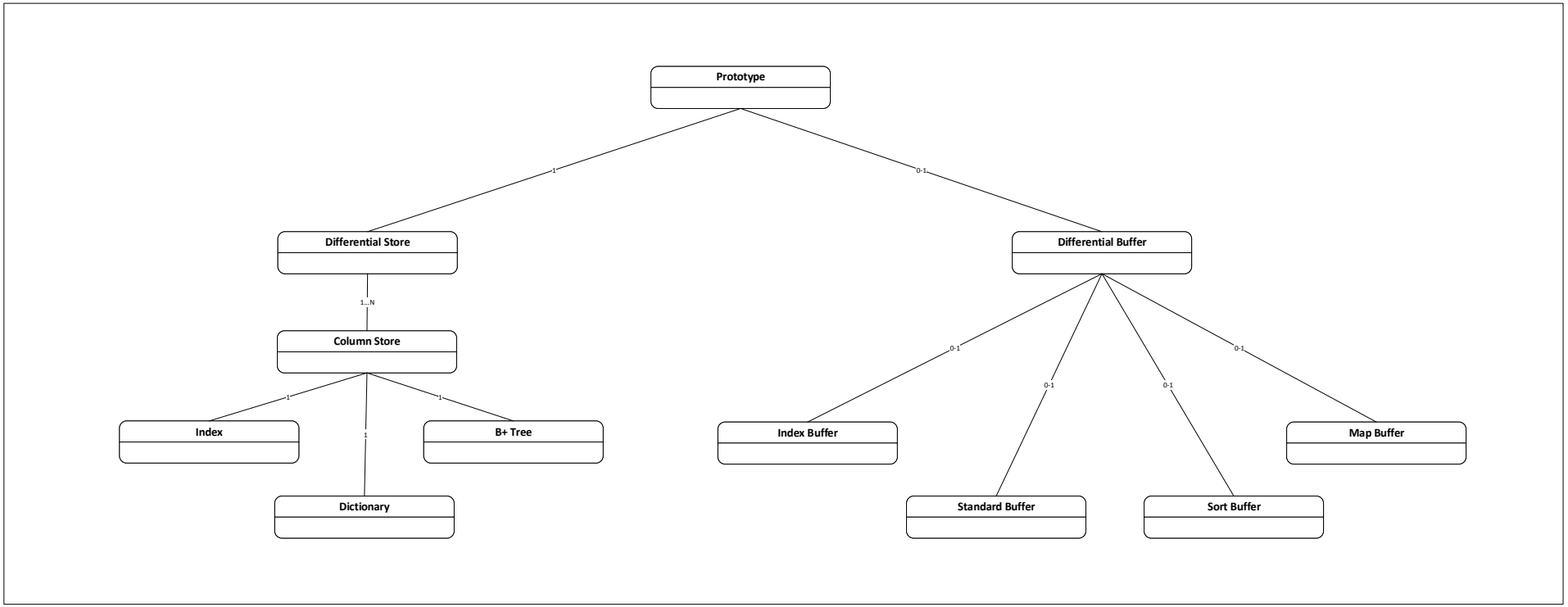


Figure 14: Diagram of prototype

5.4.Differential Store

The differential store is the most basic part of our system. It will be the point of reference to know if we are improving or not the performance. For these reason the prototype was first designed to work only using the differential store.

Since the original differential part that is used in SAP is really complex, we tried to simplify it but keeping the main functionality untouched. We avoid the use of complex compression that was depending on the data distribution (e.g. run-length encoding, sparse coding, or cluster encoding), and stick to the use of a simple dictionary encoding compression. But this would actually make the insertion process slower, so we consider it has no repercussion in the results.

The differential store will control all the different attributes of the table, each of them represented by column stores. This means that the differential store is going to select which attributes are implied and therefore only use the correspondent column stores. The column store group the functionality of the index, dictionary and tree, so the differential store only see basic operations over column store and do not need to take care of how the data is stored.

Insert operations

In order to insert a tuple in the system only through the use of the differential store, first of all, the tuple have to be divided by the attributes. Once the values of each attributes are separated, each of those values belong to a specific column store. So, the *n* value belongs to the *n* column store.

In the column store, the value is searched in the btree. If the value was already inserted, the btree will retrieve the position uses in the dictionary and then that position is added at the end of the index. In case the value is not found in the tree, means that value was not used before. So, the value is inserted at the end of the dictionary and in the btree. If is necessary the btree will be balanced. Once the process in the dictionary and in the btree is done, the column store adds the position at the end of the index.

Read operations

Regarding to the read operations, first of all, as in the write operations, the differential store has to know which attributes are involved. For the three read operations (*select, scan* and *range*) the differential store will be specified which attribute, therefore which columns are involved.

In the column store the read operations works in the following way. First, in the select operation, the column store has to find the position corresponding in the dictionary to the value that is being searched. Once that position is found, the index has to be scanned to find which positions in the index have the same value. With those positions, the column store will reconstruct the whole tuples, because all the attributes of the same tuple have the same position in all the indexes of the column stores.

The range operation works in a very similar way, since all the values in the leaf nodes in the tree are sorted, the column store need to find the minimum and the maximum value of the range. When those values are found in the tree, all the values that are in the "*leafs*" between are valid. So we need to scan the index and find the positions of the values that are in the range set. Once

we have those positions, the column store reconstruct the tuples as it does in the select operation.

The scan operation is the read operation that differs more from the rest. Here for each tuple it will reconstruct its value but only for the selected attributes. This process is done for all the tuples in the table.

5.5.Differential Buffer

The differential buffer is the part where the research and improvement of the project is going to happen around. The buffer will be an optional part of the system, since the idea is to compare the performance between the use of the buffer and the utilization of only the differential store. The differential buffer will control the different buffers, since there is one for each attribute. This is very similar to what happens with the column store, that there is one specific for each attribute of the table. The differential buffer groups the functionality of the set of the different buffers, coordinating the communication between them.

One of the goals of this research is to find the strategy of the buffer that fits better to the system. In order to find that, multiple approaches have been developed and studied, those approaches are explained into detail in the following sections. The differences between the strategies are mainly based in the insertion and merge process, the read processes remain in most of the cases very similar to each other.

5.5.1. Index Buffer

The index buffer is the most basic approach for the system. This type of buffer looks forward the idea of have the minimum repercussion into the system. The main structure of the index buffer is shown in the figure 15. This buffer consists only in one index, and takes advantage of the dictionary that is already in the differential store. This allows to use compression, with the corresponding reduce of the size of the data in the buffer.

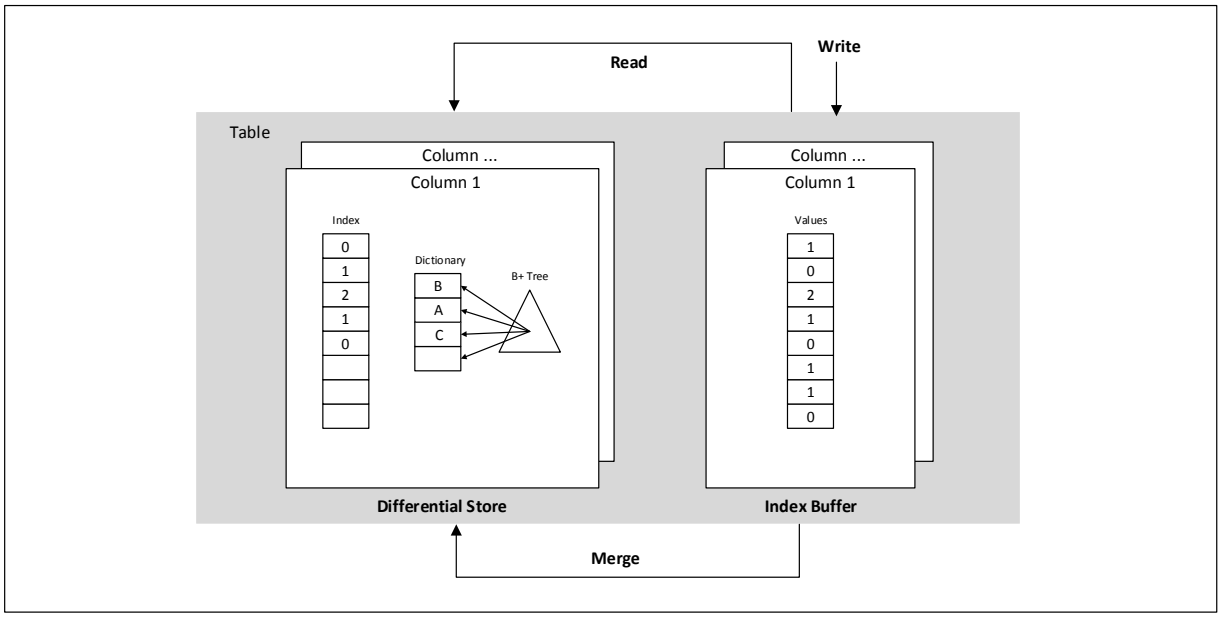


Figure 15: Index Buffer architecture

Regarding to the read operations, they work exactly in the same way than in the differential store. This is because the functionality of the column in the system is the same, just have the index divided in two different parts.

Insert operations

The insertion of data in this type of buffer is the following. First of all, the idea is to insert directly the values into the dictionary of the differential buffer, so for each insertion the system has to locate the value into the tree and the dictionary, and if necessary the value will be added. Once we have the position in the differential store's dictionary, we insert it at the end of the index that is located in the buffer.

$$\mathrm{N} = \text{Number of insertions}$$
$$\mathrm{B} = \text{Search/Insertion in the tree}$$
$$\mathrm{I} = \text{Insertion in the buffer}$$

Complexity:

$$O(N \times (B + I))$$

Merge process

Since the values are already compressed using the dictionary in the differential store, the merge process in this type of buffer has very little repercussion. The system only has to append the index of the buffer at the end of the index in the differential store.

$$\mathrm{D} = \text{number of elements in the buffer}$$
$$\mathrm{I} = \text{Insertion in the index}$$

Complexity:

$$O(D \times I)$$

5.5.2. Standard Buffer

The standard buffer continues with the concept of having a buffer with very small effect in the system. The structure of the buffer is shown in the figure 16. This buffer gets rid of the dictionary compression and focus in having all the data inserted uncompressed. This is expected to improve the insertion performance, but in contrast is going to use more space than the index buffer. Although this is will not be a problem since the buffer is going to be very small in compare to the rest of the system.

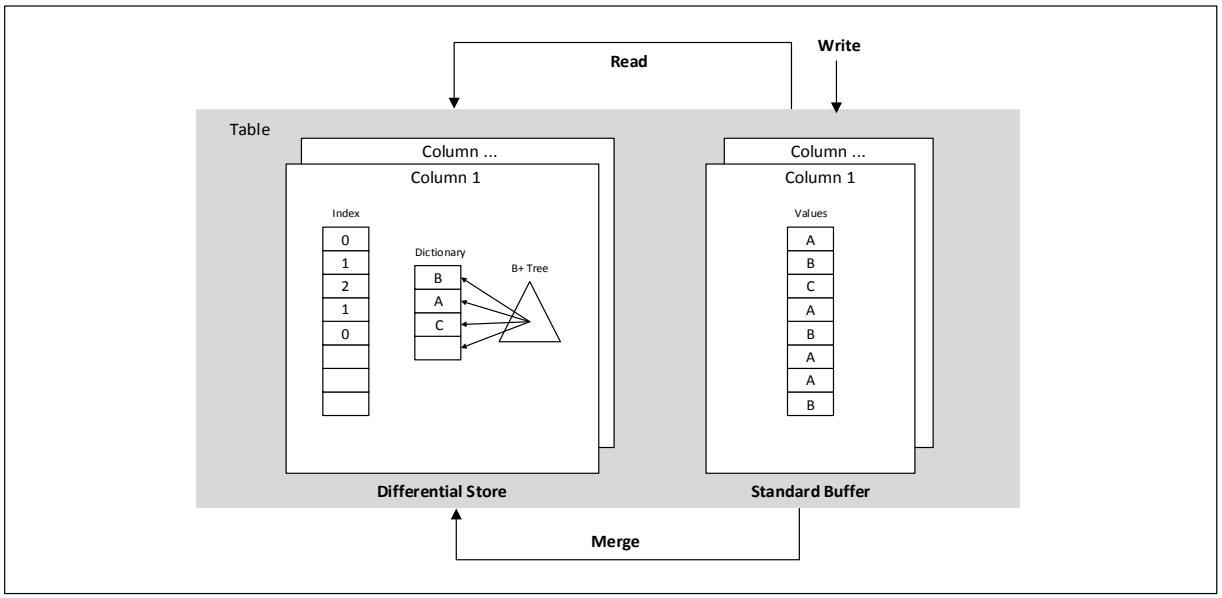


Figure 16: Standard Buffer Architecture

Regarding to the read operations, since the data is uncompressed, the process differs a little from the index buffer. Now, for the read operations, each value stored is going to be accessed in order to make sure if it matches the condition. So, in conclusion, for each read operation the system will go through all the structure where there are the values.

Insert operations

The insertion of data in the standard buffer is very simple. For each tuple that wants to be saved, first, the system divided it into the different attributes. Then, each value of each attributes is added into a list that is kept in the buffer.

$$\mathrm{N} = \text{Number of insertions}$$
$$\mathrm{I} = \text{Insertion in the buffer}$$

$$\underline{\text{Complexity}}$$
$$O(N \times I)$$

Merge process

The merge process for the standard buffer is very similar to the insertion in the differential store without the buffer. For each value in each attribute there is a search in the tree. If the value is not there, the system adds in the dictionary, updates the tree and keeps the position in the dictionary. Otherwise, if the value is already there, only keeps its position. Once the position

that belongs to the value is found, the systems adds it at the end of the index in the differential buffer.

$$D = \text{number of elements in the buffer}$$
$$B = \text{Search/Insertion in the tree}$$
$$I = \text{Insertion in the index}$$

$$\underline{\text{Complexity}}$$
$$O(D \times (B + I))$$

5.5.3. Sort Buffer

The sort buffer has a more complex background. The main concept of the buffer is sort the data in the merge process, before commit it to the differential store. The idea beyond sorting the data is based in the bulk loading techniques [30]. And especially in the advantages showed in [31] by Riku Saikkonen and Eljas Soisalon-Soininen. They explain into detail the advantages of combining local insertion sort with bulk-insertion methods and the reduction of comparisons in the tree.

The architecture of this buffer is showed in the figure 17. The concept is very similar to the standard buffer, since it has all the data inserted uncompressed. But in addition, the buffer keeps the initial position of each value within the buffer. This is going to be used to keep consistency within the attributes in the merge process.

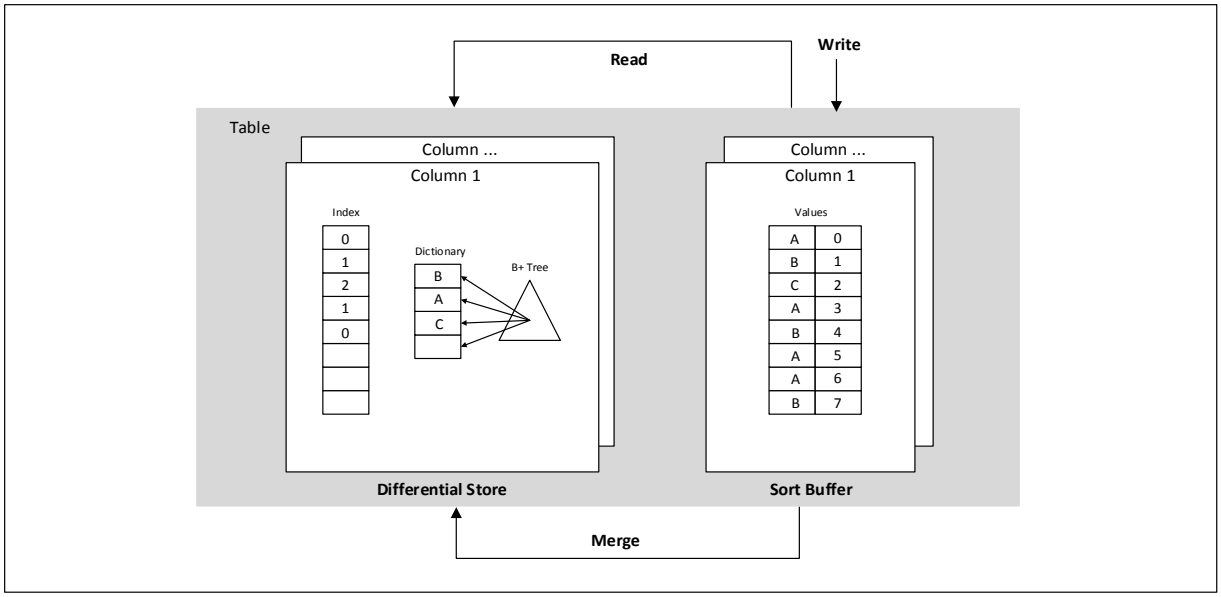


Figure 17: Sort Buffer architecture

Respect the read operations, since the data is uncompressed as in the standard buffer, for the read operations, each value stored is going to be accessed in order to make sure if it matches the condition. So, in conclusion, it is expected to have a read performance very similar to the

standard buffer, probably slightly worse because of the size of the set <data, position> is bigger than only the data.

Insert operations

The insertion in the sort buffer is very similar to the standard buffer. As it can be appreciate in the figure 17, the only difference is that the system keeps track of the position of each insertion. In conclusion, for each tuple, the system divide it in the different values of the attributes. Once it has each attribute separated, it inserts in the buffer a pair that consists in <value, position>, being the *value* the current attribute and the *position* the place that uses in the buffer.

$$\mathrm{N} = \text{Number of insertions}$$
$$\mathrm{I} = \text{Insertion of pair in the buffer}$$

$$\underline{\text{Complexity}}$$
$$O(N \times I)$$

Merge process

The merge process of the sort buffer, looks forward reduce the impact of this operation as much as possible through the sort of data. There are two reasons that motivates this approach: first, the knowledge of one of the problems of the system before the implementation of the buffer, explained into the HANA chapter, is the decrease of performance in relation with the size of the dictionary. And second, the use of dictionary compression techniques in the HANA system is because the data redundancy for most of the systems is a fact. Having this in mind, the possibility to reduce the access to the tree is, at least, something worth trying it.

This process is shown in detail in the figure 18. First of all, once the merge process is triggered, the first thing to occur in the system is the sorting of data by the value. It is very important to keep track of the original order of insertion, so when the system sorts by the value the position is also reallocated. This is the only way to keep consistency between the attributes of a same tuple. Once the values are sorted, it can be appreciated the creation of sets called *buckets*, that group the same values. The creation of the index, which is going to be append at the end of the index in the differential buffer, goes the next way. The system looks up into the dictionary of the differential buffer the value of each bucket. Once has the correspondent position for the value of the bucket ready, it inserts it into the new index buffer, but only in the positions indicated by the second value in the pair *<value, position>.* It repeats the operation for every value in the bucket, but it inserts directly the position into the new index, since because is the same value there is no need of looking up in the dictionary the same value. Repeating the process through all the buckets, will create an index of the values with the order of arrival into the buffer. And at the end, the system will only need to append this index at the end of the index in the differential store.

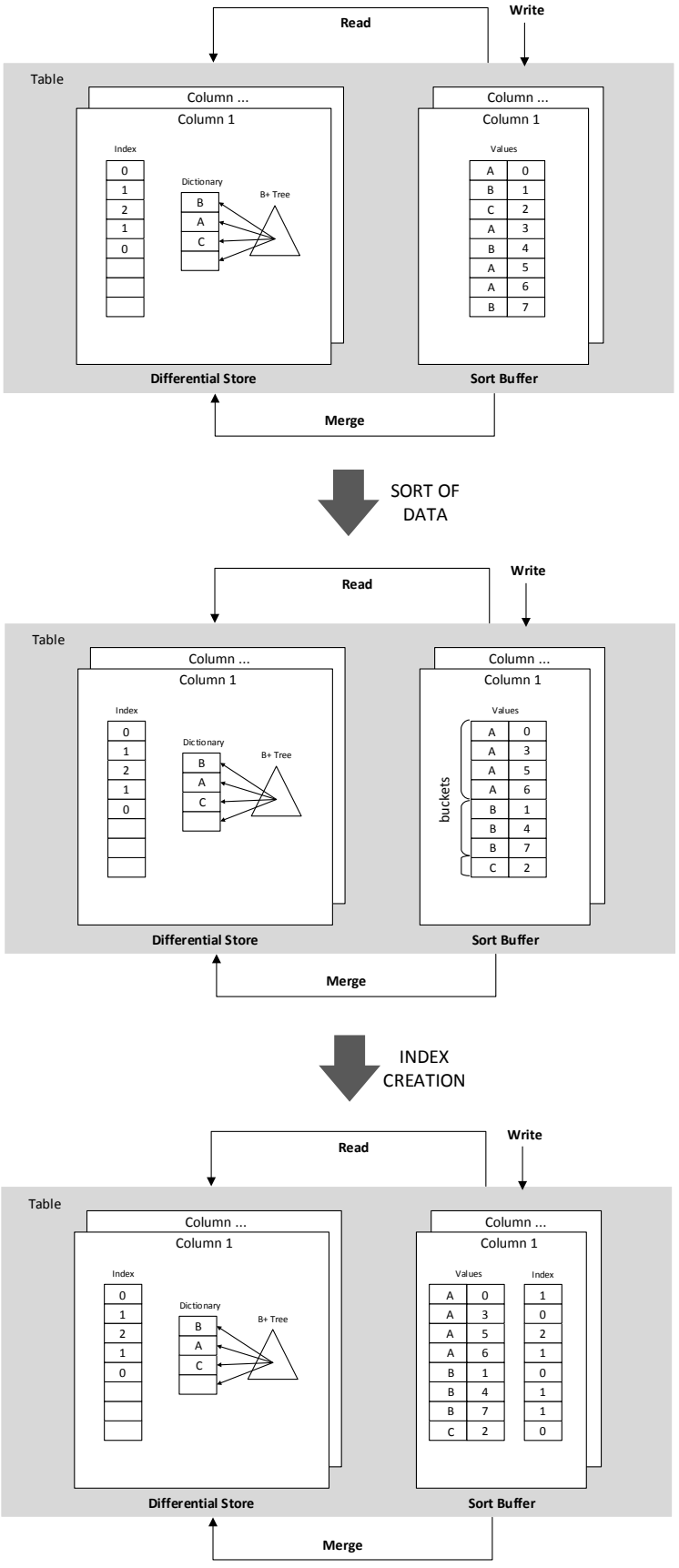


Figure 18: Merge process for Sort Buffer

The main advantage of this approach is there is no need to look up for the same value multiple times into the tree. This avoids the main problem of the differential store, going through the tree for every value insertion.

In the other hand, it is possible to do not be very optimal if the number of buckets is the same than the number of values inserted into de buffer, what means that every value is different. But this is very unlikely because this means the data distribution would not recommend to even use dictionary compression. And in this case, in the real product does not use dictionary compression, hence it does not use a tree to index the dictionary.

$$D = \text{Number of elements in the buffer}$$
$$B = \text{Search/Insertion in the tree}$$
$$I = \text{Insertion in the index}$$
$$m_j = \text{Number of buckets}$$
$$1 \leq j \leq D$$
$$r_{ij} = \text{Repetitions in each } m_j$$
$$1 \leq i \leq \#(m_j)$$
$$\sum_j \#(m_j) = D$$
$$S = \text{Sort of elements in the buffer}$$
$$O(D \times \log D)$$

Complexity:

$$O\left(\left(\#(m_j) \times B\right) + (D \times I) + (D \times I) + S\right)$$

5.5.4. Map Buffer

The map buffer is based in the concepts explained in the sort buffer. The main idea behind is to reduce the accesses to the dictionary, and hence the tree, in the differential store. The buffer is very similar to the standard buffer, but with an improved merge process with the idea of reduce the impact of the operation.

The architecture of this buffer is showed in the figure 18. The concept is very similar to the standard buffer, since all the data inserted is uncompressed. But in addition, the buffer has an additional structure, a map that will be filling in the merge operation in order to avoid duplicate searches in the dictionary of the differential store.

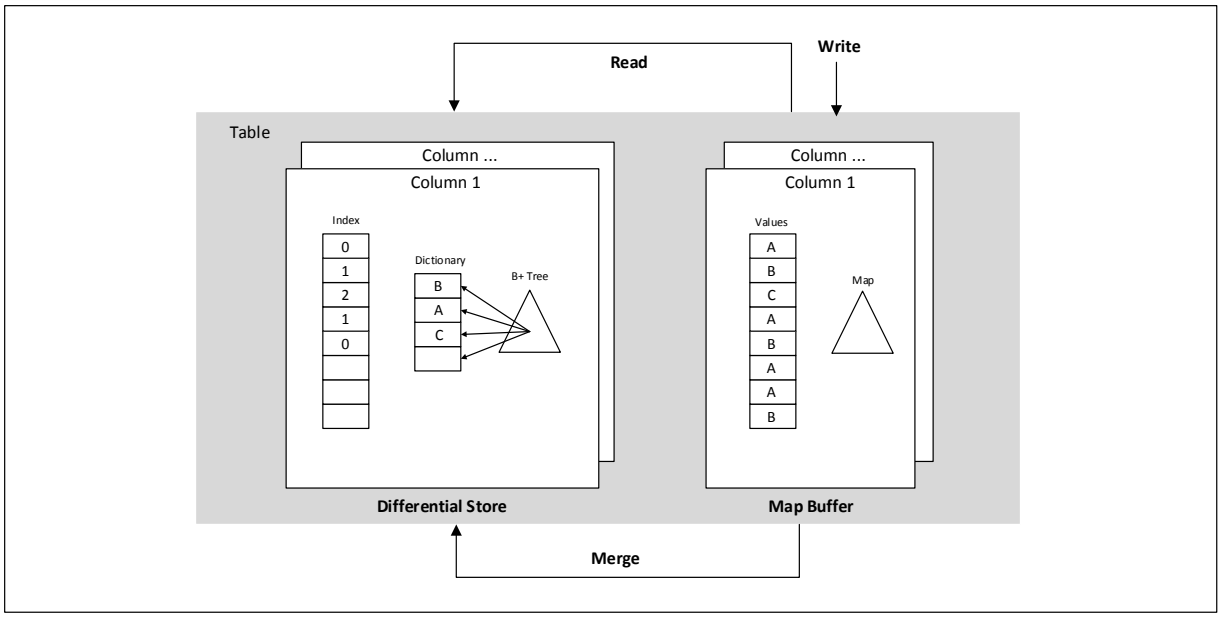


Figure 19: Map Buffer architecture

Respect the read operations, since the data is uncompressed as in the standard buffer, for the read operations, each value stored is going to be accessed in order to make sure if it matches the condition.

Insert operations

The insertion in the buffer is exactly the same than in the standard buffer. For each tuple, the system splits it into the different attributes. And then for each value, of each attribute, is store in the buffer.

$$\mathrm{N} = \text{Number of insertions}$$
$$\mathrm{I} = \text{Insertion in the buffer}$$

$$\underline{\text{Complexity}}$$
$$O(N \times I)$$

Merge process

The merge process is what difference the map buffer and the other buffers. The motivation of this buffer is the same than for the sort buffer, to reduce the impact of the merge. In order to achieve that, the buffer will use an additional structure, a map to avoid look up into the dictionary of the differential store for the same values multiple times.

During the merge operation, for each value, the system looks it up into the map. If the value is not found there, is because is the first time it appears in the merge operation, so the system searches it in the dictionary of the differential store. Once it has the correspondent position, it adds it at the end of the index in the differential buffer and, at the same time, it adds it in the map of the buffer. Doing this, if a value is repeated in the buffer, instead of looking it up in the

dictionary of the differential store, which can be really large, the system searches it in the map of the buffer.

D = number of elements in the buffer
R = Repeated values in the buffer
nR = Non-repeated values in the buffer
B = Search/Insertion in the differential store's tree
M = Search/Insertion in the differential buffer map
I = Insertion in the index

Complexity

$$O((D \times M) + (nE \times B) + (D + I))$$

6. Implementation of the prototype

This chapter is going to explain the implementation of the main parts of the prototype. The first part of the section is going to review the implementation of the basic part of the architecture, the differential store. About the differential store, it is going to be explained how the four operations are implemented. After, the chapter focus in the differential buffer. Exactly in the implementation of the insertion and the merge operations, because the read operations are very similar to each other.

An important aspect is that this chapter focus on the implementation of the operations that will be evaluated in the next chapter. This section will provide the knowledge to understand how the different operations were implemented and how they run in the differential store and the differential buffer. The chapter focus in the prototype, although there are other parts that influence in the code. It is considered that those parts lack of importance over the rest of the implementation. Some examples of those parts are: the class that loads the csv files, the class that creates the csv files following the specifications of entropy, the class that generates random data for each field, etc... In total the source code consists in more than 20 different classes. Since the topic here is to study the performance of the differential store and the differential buffer, the details of the rest of the implementations will be left out of this document.

Also, with the objective of keep the read of the document as agile as possible, it will not do a review of each method of each class of the prototype. The names of the variables, methods and classes try to be as understable as possible, in addition the source code is intensibly commented. A diagraman of the all the classes and methods that belongs to the prototype can be found in the Figure 20.

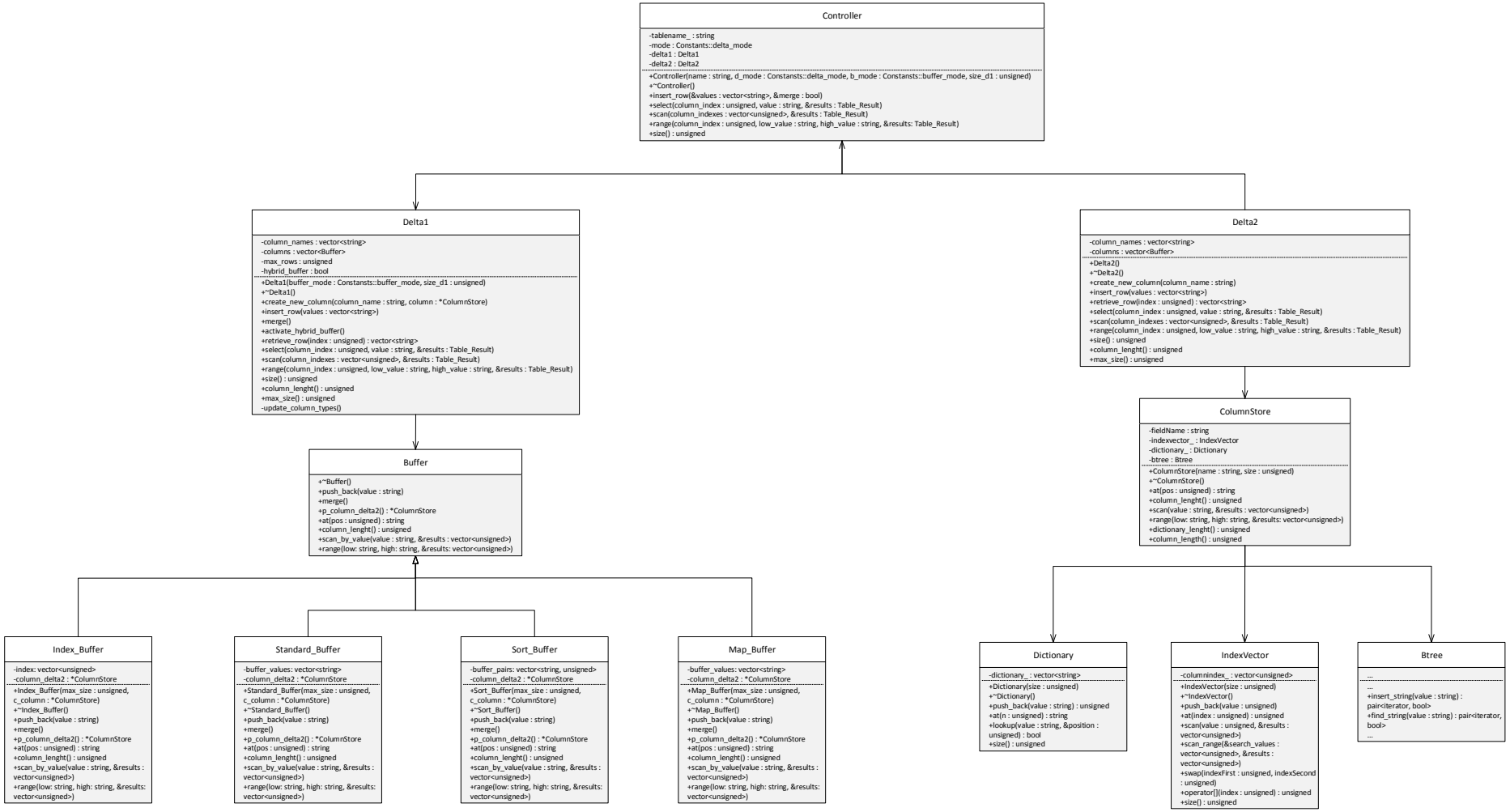


Figure 20: Class diagram of the prototype

6.1.Controller

The *Controller* class represents a table. This class consists in a *Delta2* and an optional *Delta1*. The *Controller* will be in charge of execute the different operations and combine the results of the *Delta2* and *Delta1*, whenever is necessary. It also allows to execute the buffer in two different modes. One allows to run *select/scan/range* operations in the buffer, the other mode always merge the buffer before execute one of those operations.

6.2.Differential Store

The differential store is known in the implementation as *Delta2*. The differential store is mostly a table itself, it is a set of columns. Those columns of the table are known as *ColumnStores*. As it will be explained in the following operations, *Delta2* controls the communication between the columns. Most of the times it search the desired rows in the selected column, and then reconstruct those records with the desired attributes.

And remarkable aspect here was the implementation of the tree structure. Our structure is based in the implementation of Timo Bingmann of "Memory based B+ tree implementation as C++ template library" under the GNU General Public License v3 (GPLv3).

Insert operation

Pseudocode of insert:

```
void Delta2::insert_row(const vector<string> &values)

      For every value in the record

            void ColumnStore::push_back(const std::string &value)

                  const_iterator Btree::insert_string(const
                  std::string &key)

                        void IndexVector::push_back(unsigned
                        &value)

                              Insert the value returned from the tree
                              into the index

      End-For
End
```

The method insert in *Delta2* is called from *Controller*. This method will insert a record, readed from a file or randomly generated, in the system. The records will be in a row format, so it has to iterate all the positions in order to access to each attribute. For each attribute, it will call push_back from the corresponding *ColumnStore*. In the *ColumnStore*, it will call insert_string from *Btree.* This method will go through all the tree looking for the desired value, and optimization was used here. Instead of look up for the value, and if it was not found, insert it. It

goes through the path it would use for the insertion, if at the end the value is already there, it returns it. Otherwise it inserts it in the position, balance the tree, insert it at the end of the *Dictionary* and return it. After it has the corresponding position of the value in the *Dictionary*, it uses the push_back method from *IndexVector* for insert that position at the end of the Index. This procedure will be repeated for each attribute, until the whole row is inserted.

Select operation

Pseudocode of select:

```
void Delta2::select(const unsigned column_index, const
std::string &value, Table_Result &results)

      void ColumnStore::scan(const std::string &value,
      std::vector<unsigned> &results)

            const_iterator Btree::find_string(const std::string
            &key)

            If value was found then

                  void IndexVector::scan(const unsigned value,
                  std::vector<unsigned> &results);

            End-If

      For every position returned from scan

            vector<string> Delta2::retrieve_row(const unsigned
            index)

      End-For

End
```

First of all, the *select* method in *Delta2* is called from the C*ontroller.* This method executes the select in two steps.

The first step is to find the rows that match the condition. For this reason, from *delta2* the system calls the *scan_by_value* method in the *ColumnStore* of the corresponding column. Since *delta2* has a pointer to multiple *ColumnStore*, one for each column in the table, it has to search in the designed column. The *scan_by_value* method will look up in the tree for the value, using the *find_string* method from the Btree, which returns the position in the dictionary of the value if the value was already there and in case it was found, it will search which rows matches that position of the dictionary, through the use of *scan* in the *IndexVector* class.

The second step takes place once it is known the rows that are valid. The system knows the positions of the rows in the table, so now, using the *retrieve_row* call, it will reconstruct the records with all the attributes in a row format, and store it in the result table. In order to reconstruct the values, it is essential to use the at method of the *Dictionary*, using the value of the *IndexVector* as parameter.

It has to take in consideration that both phases executes suposing there are rows with that value in that column. In case there is not, it will simply not find any record that matches the condition and it will not reconstruct any row.

Scan operation

Pseudocode of scan:

```
void Delta2::scan(const std::vector<unsigned> &column_indexes,
Table_Result &results)

      For every row in Delta2

            Create a new row

            For every attribute requested

                  Access to the real value of the attribute
                  Add it to the new row

            End-For

            Add it to the table of results

      End-For

End
```

The *scan* method in *delta2* is called from the *controller*. This method will go through all the rows, and will reconstruct each one of them but only with the selected attributes in the parameter of the function. In order to access to the real value of each position in the *IndexVector*, it will use the method *at* of the class *Dictionary* using as a parameter the value of the Index. This is necessary because of the dictionary compression. So for access each value, it will takes two access to two different vectors, the index and the dictionary. Once every row has the right attributes, it will be added to the table of results.

Range operation

Pseudocode of range:

```
void Delta2::range(const unsigned column_index, const
std::string &low_value, const std::string &high_value,
Table_Result &results)

      If the low_value and the high_value are the same

            Executes a select of any of them

      Else
            void ColumnStore::range(const std::string &low, const
            std::string &high, std::vector<unsigned> &results)

                  Find the first value in the dictionary using the
                  tree

                  Find the second value in the dictionary using
                  the tree

                  For every value in between

                        Add it to a vector of desired values

                  End-For

                  Sort the vector of values

                  void
                  IndexVector::scan_range(std::vector<unsigned>
                  &search_values, std::vector<unsigned> &results)

                        For every row in the column

                              Binary Search in the vector of
                              search_values

                              If the value matches then add the
                              position to results

                        End-For

                  End

            End

      For every position returned from range

            vector<string> Delta2::retrieve_row(const unsigned
            index)

      End-For

End
```

The range method in *Delta2* is called from *Controller*. The first step of all is to make sure both values, the low and the high of the range are the same, otherwise it will execute a select by that value. After the operation will execute in two different phases.

First, it has to find the valid rows in the table. In order to achieve that, the system calls the range method of the *ColumnStore.* The column is set by the parameter. This method will find every valid position in the dictionary using the tree. Then it will sort all those positions and call scan_range in the *IndexVector*. In this other method, the system will compare compressed value with the valid positions of the dictionary. In case it matches, it will save the position.

Once it has every valid position within the index, the range method in *Delta2* will reconstruct those rows. In order to do this, it will decompress every attribute needed.

6.3.Differential Buffer

The differential buffer is known in the implementation as *Delta1*. The differential buffer, as it happens with the differential store, acts as a table itself because it is a set of multiple columns. The columns can implement one of the four approaches: *Index Buffer, Standard Buffer, Sort Buffer* and *Map Buffer.* All those approaches inherit from the abstract class named *Buffer*. This allows to *Delta1* to handle a set of *Buffers* through the use of polymorphism without knowing what type of buffer is using. It also allows the use of a combination of buffers since the system does not need to know which one is using.

6.3.1. Index Buffer

Pseudocode of insertion:

```
void Delta1::insert_row(const vector<string> &values)

      If the current size of the buffer + 1 is bigger than the
      max_size

            For every column

                  void Buffer::merge()

            End-For

      End-if

      For every value in the record

            void Buffer::push_back(const std::string &value)

                  const_iterator Btree::insert_string(const
                  std::string &key)
```

```
                    Insert the value returned from the tree into
                    the index

                    void IndexVector::push_back(unsigned
                    &value)

      End-For
End
```

The insertion in the *Index_Buffer* is mostly the same than for the Differential Store. The function in *Delta1* is called from the *Controller.* And calls the push_back method from the corresponding column using *Buffer*. After it does exactly the same than in the Differential Store. It calls insert_string from *Btree.* This method will go through all the tree looking for the desired value, and optimization was used here. Instead of look up for the value, and if it was not found, insert it. It goes through the path it would use for the insertion, if at the end the value is already there, it returns it. Otherwise it inserts it in the position, balance the tree, insert it at the end of the *Dictionary* and return it. After it has the corresponding position of the value in the *Dictionary*, it adds it at the end of the index from *Index_Buffer*.

Pseudocode of merge:

```
void Index_Buffer::merge()

      For every value in the index

            Using the pointer to the corresponding ColumnStore...
            void IndexVector::push_back(unsigned &value)

      End-For

      Empty the index

End
```

The merge in the *Index_Buffer* is called from *Delta1*. When it detects that the next insertion is going to surpass the maximun size of the buffer, it produces the merge. In this type of buffer, since all the values in the buffer already went through the dictionary compression, it only needs to access to the push_back method from the *Index_Vector*. This is done using the pointer to the *ColumnStore* that points to the column from the Differential Store that belongs to this column of the buffer.

Respect the select, the scan, and the range methods those executes exactly the same than the *ColumnStore* in the Differential Store section above. Because it uses the dictionary compression from the differential store, the only difference is that instead of access to the *Index_Vector*, it access to the *Index_Buffer*.

6.3.2. Standard Buffer

Pseudocode of insertion:

```
void Delta1::insert_row(const vector<string> &values)

      If the current size of the buffer + 1 is bigger than the
      max_size

            For every column

                  void Buffer::merge()

            End-For

      End-if

      For every value in the record

            void Buffer::push_back(const std::string &value)

                  Add value at the end of buffer_values
      End-For
End
```

The insertion in the *Standard_Buffer* is the most basic approach. The function in *Delta1* is called from the *Controller*. And uses the push_back method from the abstract class *Buffer*. Through the use of polymorphism, the push_back method of the *Standard_Buffer* is used. This method adds the value at the end of a vector called *buffer_values* that is kept in the buffer.

Pseudocode of merge:

```
void Standard_Buffer::merge()

      For every value in the buffer_values

            Using the pointer to the corresponding ColumnStore...

            const_iterator Btree::insert_string(const std::string
            &key)

            Insert the value returned from the tree into the index
            void IndexVector::push_back(unsigned &value)


      End-For

      Empty the buffer_values

End
```

The merge in the *Standard_Buffer* is a function called from *Delta1.* When *Delta1* detects that the next insertion is going to overflow the buffer, it triggers the merge. This type of buffer does the opposite that the *Index_Buffer*. In the *Standard_Buffer* the dictionary compression of the values is done in the merge process. For each value, it calls insert_string from *Btree.* This method will go through all the tree looking for the desired value, and optimization was used here. Instead of look up for the value, and if it was not found, insert it. It goes through the path it would use for the insertion, if at the end the value is already there, it returns it. Otherwise it inserts it in the position, balance the tree, insert it at the end of the *Dictionary* and return it. After it has the corresponding position of the value in the *Dictionary*, it adds it at the end of the index from *Index_Buffer*.

Respect the select, the scan, and the range methods, follow a similar pattern than in the *ColumnStore*. For the select, the scan_by_value method in the *Standard_Buffer* finds out what positions from have the same value than the value the system is looking for. But instead of decompress each value, as it does the *ColumnStore*, it compares it directly. Once all the valid positions are found, *Delta1* reconstruct the selected rows. About the scan, *Delta1* reconstruct all the rows with the selected attributes. Since all the values are uncompressed, it access to the value in *buffer_values* directly. And for the range, to find out what rows match the condition of the set, it uses the range method in the *Standard_Buffer*. This method is the same than the scan_by_value, but instead of compare it to one value, it checks if is smaller than the high value of the set and bigger than the small value of the set. Also after the selected rows are found, *Delta1* is in charge of reconstruct the rows.

6.3.3. Sort Buffer

Pseudocode of insertion:

```
void Delta1::insert_row(const vector<string> &values)

    If the current size of the buffer + 1 is bigger than the
    max_size

        For every column

            void Buffer::merge()

        End-For

    End-if

    For every value in the record

        void Buffer::push_back(const std::string &value)

            Creates a pair, the first position is the value
            and the second the current number of rows in the
            buffer

            Add the pair at the end of buffer_pairs
```

```
    End-For
End
```

The insertion in the *Sort_Buffer* is very similar to the insertion in the *Standard_Buffer*. The only difference is that instead of only keeping the uncompressed values, it also has the original position the value had in the buffer. The insertion is called, as in the other strategies, in the *Delta1* which is used in the *Controller*. *Delta1* uses the push_back method from the abstract class *Buffer*. This ends calling the push_back method in the *Sort_Buffer.* This method creates a pair with the value that wants to be inserted and the position that will take up in the buffer. That pair is inserted in the *buffer_pairs* that is kept in the buffer.

Pseudocode of merge:

```
void Sort_Buffer::merge()

    Sort the buffer_pairs using the values of the first position

    Current_value = first value of the buffer
    Previous_value = first value of the buffer
    Create a new_index with the size of the buffer_pairs

    Using the pointer to the corresponding ColumnStore...

    const_iterator Btree::insert_string(const std::string &key)

    Insert the position returned from the tree in the new index
    in the position pointed by the second value of the pair

    Previous_pos_dic = the position returned from the tree

    For the rest of values in the buffer of pairs

        Update current value

        If the current_value is different than the
        previous_values then

            Using the pointer to the corresponding
            ColumnStore...

            const_iterator Btree::insert_string(const
            std::string &key)

            Insert the position returned from the tree in the
            new index in the position pointed by the second
            value of the pair

            Update the Previous_pos_dic

        Else

            Insert the previous_pos_dic in the new index in
            the position pointed by the second value of the
            pair
```

```
            Update previous_value to current_value

      End-For

      For every value in the new index

            Using the pointer to the corresponding ColumnStore...
            Copy the new index at the end of the index in the
            Index_Vector

      End-For

      Empty the buffer_values

End
```

The merge function of the *Sort_Buffer* is called from *Delta1*. As in the other cases, the merge process is triggered when *Delta1* detects that the next insertion is going to overflow the buffer. This buffer is based in sort the data in order to group the similar values, save look up in the dictionary and do a sorted insertion.

First of all, it sorts the data based in the first value of the pair. For this purpose the stable_sort algorithm is used, because the traditional sort algorithm has troubles sorting data with multiple repetitions. Once the data is sorted, it iterates over all the sorted values, keeping the previous value and the current value. Each of those is compared and depends of the results the merge of that value is done in a different way.

If the values are different, it calls insert_string from *Btree.* This method will go through all the tree looking for the desired value, and optimization was used here. Instead of look up for the value, and if it was not found, insert it. It goes through the path it would use for the insertion, if at the end the value is already there, it returns it. Otherwise it inserts it in the position, balance the tree, insert it at the end of the *Dictionary* and return it. After it has the corresponding position of the value in the *Dictionary*, it adds it to the *new_index* in the position specify in the second value of the pair. It also updates the *previous_value* and the *previous_pos_dic.*

But if the values are the same, it means the value that was previously prepared to merge and the current value are the same, so some optimizations can be done. It inserts the *previous_pos_dic* to the *new_index* in the position specify in the second value of the pair. And it only updates the previous value.

Once it have iterated over all the values in the buffer, the *new_index* is created. This index only needs to be added at the end of the index in the *Index_Vector* of the column specified in the pointer of *ColumnStore*.

Respect the read operations: select, scan, and range. They execute exactly in the same way than in the *Standard_Buffer*, because the data is kept uncompressed. The only difference is that instead of access to the values directly, it has to access to the first position of the pair.

6.3.4. Map Buffer

Pseudocode of insertion:

```
void Delta1::insert_row(const vector<string> &values)

      If the current size of the buffer + 1 is bigger than the
      max_size

            For every column

                  void Buffer::merge()

            End-For

      End-if

      For every value in the record

            void Buffer::push_back(const std::string &value)

                  Add value at the end of buffer_values
      End-For
End
```

The insertion in the *Map_Buffer* is exactly the same than in the *Standard_Buffer*. The function in *Delta1* is called from the *Controller*. And uses the push_back method from the abstract class *Buffer*. Through the use of polymorphism, the push_back method of the *Standard_Buffer* is used. This method adds the value at the end of a vector called *buffer_values* that is kept in the buffer.

Pseudocode of merge:

```
void Map_Buffer::merge()

      Create a map

      For every value in the buffer_values

            If the value is not found in the map then

                  Using the pointer to the corresponding
                  ColumnStore...

                  const_iterator Btree::insert_string(const
                  std::string &key)

                  Insert the value returned from the tree into the
                  index and the map
                  void IndexVector::push_back(unsigned &value)
                  std::pair<iterator, bool> Map::insert(const
                  string &value);

            Else
```

```
                Insert the value returned from the map into the
                index
                void IndexVector::push_back(unsigned &value)

        End-For

        Empty the buffer_values

End
```

The merge in the *Map_Buffer* is a function called from *Delta1*. The merge is activated when *Delta1* finds out that the next insertion is going to overflow the buffer. This buffer uses an unordered structure, in particular a red/black tree, in order to take advantage of the repeated values and the smaller size of that structure in compare to the dictionary of the differential store. At the beginning of the process it creates the empty structure, and for each value it searches it in there. If the value is not found, then it calls insert_string from *Btree.* This method will go through all the tree looking for the desired value, and optimization was used here. Instead of look up for the value, and if it was not found, insert it. It goes through the path it would use for the insertion, if at the end the value is already there, it returns it. Otherwise it inserts it in the position, balance the tree, insert it at the end of the *Dictionary* and return it. After it has the corresponding position of the value in the *Dictionary*, it adds it at the end of the index from *Index_Buffer*. And it also adds the position to the map. But if the value is found, then it only adds it at the end of the index from *Index_Buffer*, because it was already inserted in the dictionary in one of the previous lap of the loop.

Respect the read operations: select, scan, and range. They execute exactly in the same way than in the *Standard_Buffer*, because the data is kept uncompressed.

7. Evaluation

This section considers the performance of the prototype using the different approaches. The performance in different areas is going to be compared and evaluated. The main characteristics to be evaluated are: first, the insert and merge performance, and second, the performance of the read operations. This performance is compared by using some synthetic tests, which are explained at the beginning. And at the end, a real scenario is evaluated.

7.1. Test scenarios

The performance comparison of the different strategies for the prototype are done by using different test. In those test the two main aspect to be compare are going to be evaluated. The first aspect to be compare is the insertion performance into the system and its improvement, if any. The second aspect is the repercussion in the query performance, a small impact is expected. For that purposes, there are two different test available to run in the system.

The main structure of the test is shown in the Figure 21. Although there are two different characteristics to be measured, there are some parts that are common in both test. This feature is taken in advantage in the design of the tests and exploited through the use of inheritance. So, the common parts of the test are done in the *Base_Test*. Those parts are mostly related about the insertion of data. It gives the possibility of insert the data from a *comma separated values* file (csv) or to generate it randomly. If a path for the file is specified, then that file is read. But if that is not specified, then the system generates random values. It can also be told to the system how many columns create for the table in case it is generating random values, or how many columns are needed to be loaded from the file. Another part that is taken care of is the data distribution. This is done specifying the length of the dictionary and the number of total insertions. If there is a small dictionary in compare to a big index, it can be assure that there is an insignificant variety of values in that column.

One of the specific test is the *Merge_Test*. This test will be in charge of compare the performance of insert a large amount of data directly into the differential store or the time taken using the buffer, if using the buffer it will consider the time used in the insertion and the time spent merging the buffer. So mainly, shows if improvement of time inserting in the buffer pays off for the time merging the buffer.

Regarding to the other specific test, the *Query_Test*, this will measure the performance of the read operations (*select/scan/range)*. In order to ask for real data, data that was already stored, the system will select random values to be asked for. Then it will execute the designated operations, since it supports to select a specific read operation or the three of them. Here it can also be configured how many of each operations are executed for each loop of insertions.

The time measures of all the relevant data from the tests: insertion, merge, select, scan, and range; are saved in text files which values are separated using tabulations. This is for load the data into spreadsheet software and analyze it.

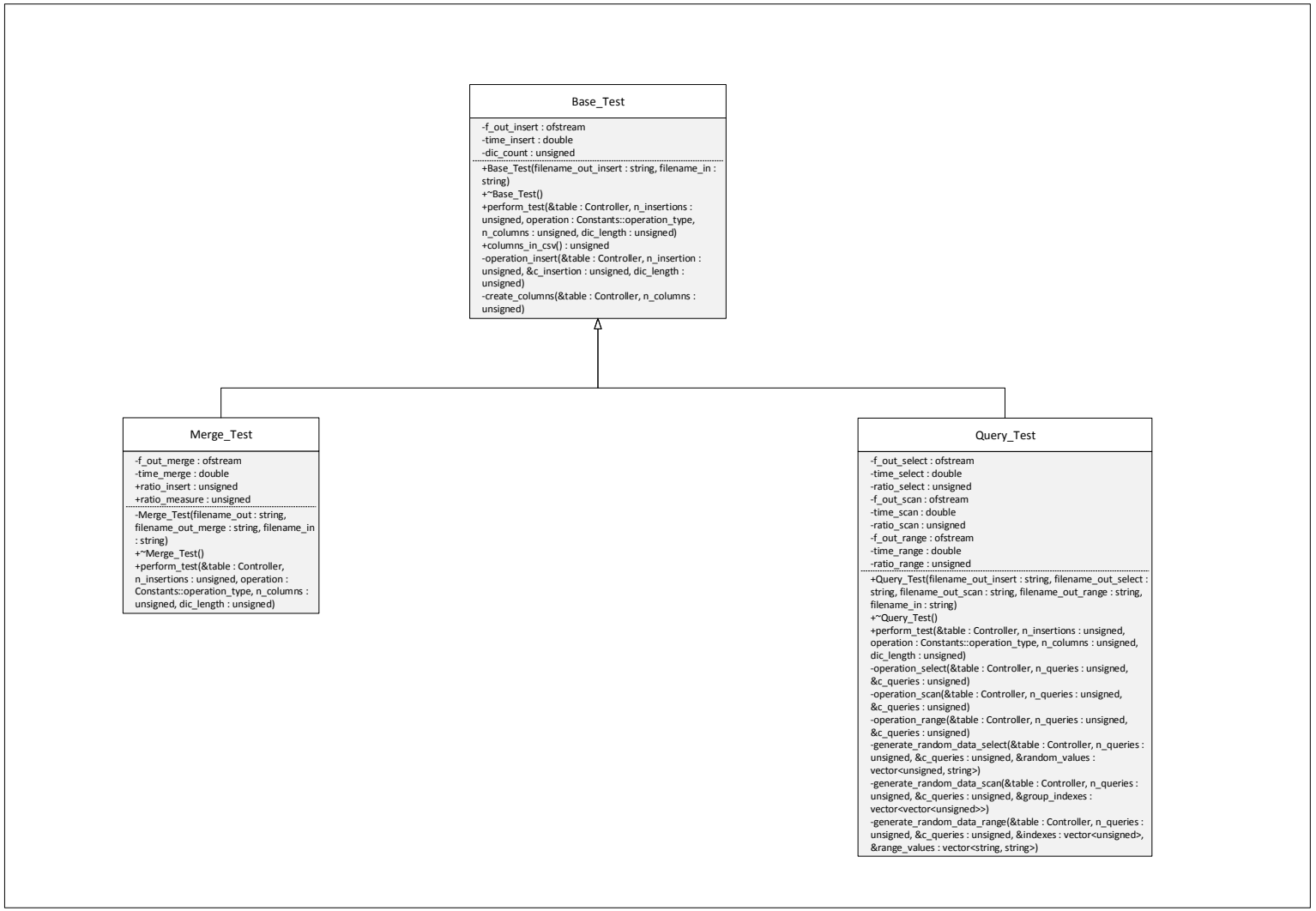


Figure 21: Class diagram for the tests

Another relevant aspect of the test are the configuration of the different parameters. The most important parameters for the tests are the data distribution, the size of the buffer (when need it), and the number of attributes involved.

Regarding to the data distribution. Since the entropy of the data is going to be decisive, the test were configured to have a big range of entropy. In order to establish this entropy, it is possible to set the number of different values that will be added to the dictionary. Once the size of the dictionary is reached, the system inserts values that are retrieved from the dictionary. This assures that the systems grows the dictionary to the desirable size, and then, only inserts repeated values. For example, if the size of the dictionary is 25, and the total number of insertions is 100, the system will have 4 times each value. The tests were configured to an initial dictionary size of 25,000 and it doubles for each tests until reaches 12,800,000. This gives 10 different samples.

About the size of the buffer, when it is active, this is also decisive of the performance. This is because the data is stored in a different way than in the differential store, hence it is expected to have a repercussion in the data insertion, but also in the read performance of those values. The size of the buffer will go from 100,000 to 12,800,000 tuples. The size of the buffer doubles for each test, this gives 8 different samples in a big range of sizes.

And in relation to the number of attributes, this aspects defines the number of columns of the table. The number of attributes in the test were kept to one, since the time of any operation has a fixed increase to the number of attributes. The reason of this is that always that the characteristics of the attributes are the same (data distribution, number of insertions, etc…) the procedure to execute each operation for each attribute is the same. So, having the two attributes will only double the operations, and having ten attributes will multiply by ten the number of exactly the same operations. Because of these reasons, the test run only with one attribute.

7.2.Insertion and Merge performance

The first aspect to be analyzed is the insertion, and hence merge, performance. There has been many different parameters to take into consideration to evaluate this aspect of the prototype.

One decisive aspect to the test is the number of insertions and the frequency of measuring. Since the research it is related to big data, the most recommendable was to have a huge amount of data to be inserted. It is considered that insert 128 million of tuples is big enough to test the system. Regarding to the frequency of measuring, it has to bight enough to be representative and ignore small fluctuations, and at the same time small enough to allow multiple samples of the total insertions. It was considered that 10 samples from the 128 million was a good fraction, so the system measures every 12.8 million.

In the following sections each of the different strategies used for the prototype are going to be analyzed. For each one, the performance according to the different parameters is going to be studied and explained. The assumptions are based in the data that can be found in the index of

tables. In order to facilitate the read and comprehension of the different evaluations, only resuming graphics are going to be showed.

7.2.1. Buffer disabled

In the Figure 22, it can be appreciated how the time of insertions evolves while the system continues inserting data. The longest phase is the first, for all the data distribution, this is because at the beginning there are no values either in the dictionary or in the tree. So, until the dictionary and the tree reaches the desire size specified in the test, the tree will be growing and balancing what is the most expensive operation in the insertion process. The more different values we have, the more time this phase will take, as it can be appreciated in the graphic. But once all the values are in the system, the time of insertions becomes constant, because all the data is repeated uniformly, so the process only needs to find the value in the dictionary and add it to the index. Since the test measures every 12.8 million of insertions and the dictionary size is 25,000, what indicates the number of different values, this means it will search in the tree and insert in the index 512 times each different value. For 50,000 different values it will do it 256 times. And so on, until 12,800,000 different values where it will only do it one time for each value. How this affects to the insertion time is showed in the Figure 23. Since in the case of 12,800,000 different values the tree will be the biggest, it will be the worst case and hence the slowest.

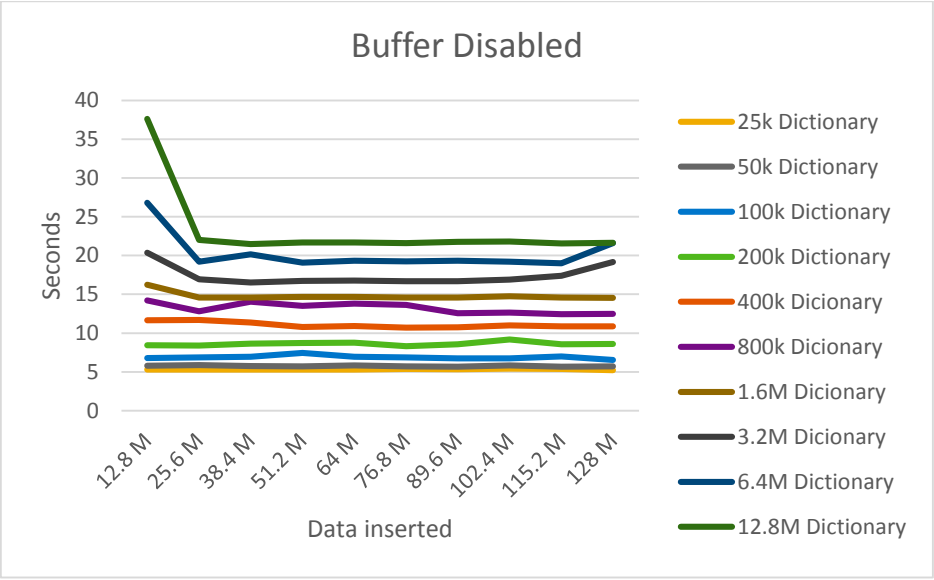


Figure 22: Insertion times respect data already inserted in Buffer Disabled

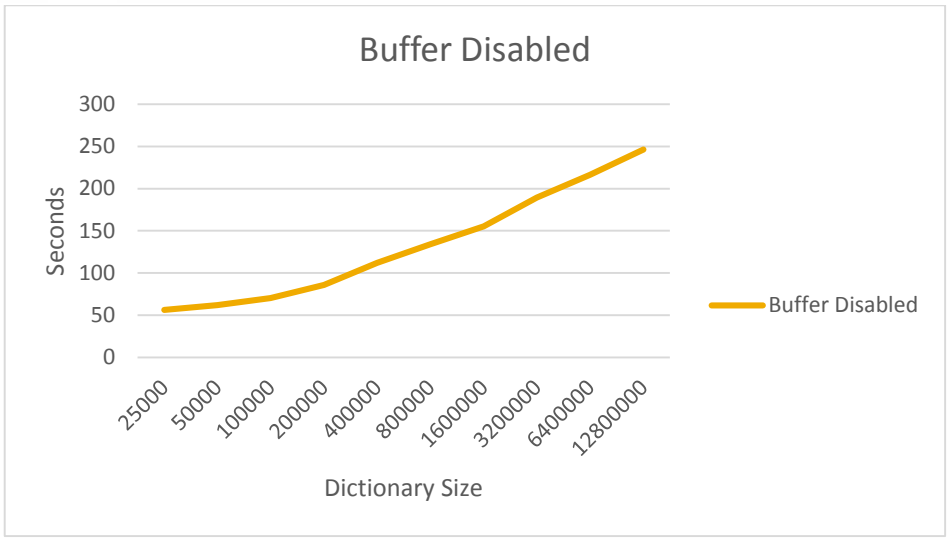


Figure 23: Insertion times respect data distribution in Buffer Disabled

7.2.2. Index Buffer

The index buffer is the simplest approach for the buffer. The Figure 24 shows the insertion times for the different data distribution for the smallest buffer configuration, 100,000 values. As it can be appreciated, the behavior is basically the same than when the buffer is disabled, this is because as it can be appreciated in the design chapter it does mostly the same that when the system has the buffer disabled. The longest part is the beginning, when it generates the tree with the values, after that the insertion times are more or less constants. It can be appreciated some variations, these are attached to the system load at the moment. Respect to the Figure 25, it shows it does not matter the size of the buffer. The time of insertion is mostly the same for a buffer of 100,000 values than for a buffer of 12,800,000 values. The explanation of this is because is the same execute 128 merges using a buffer of 100,000 values that execute one merge using a buffer of 12,800,000 values.

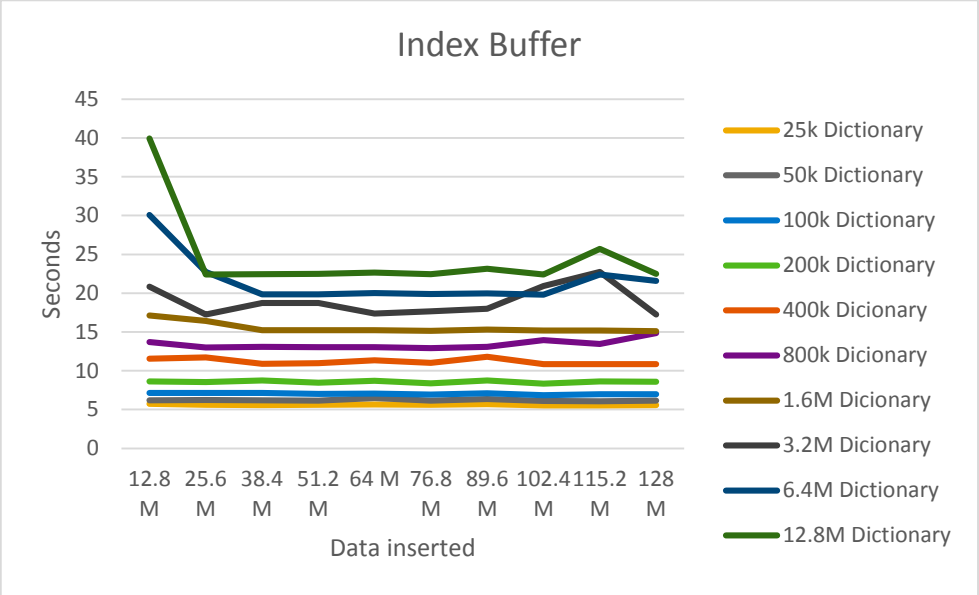


Figure 24: Insertion times respect data already inserted in Index Buffer

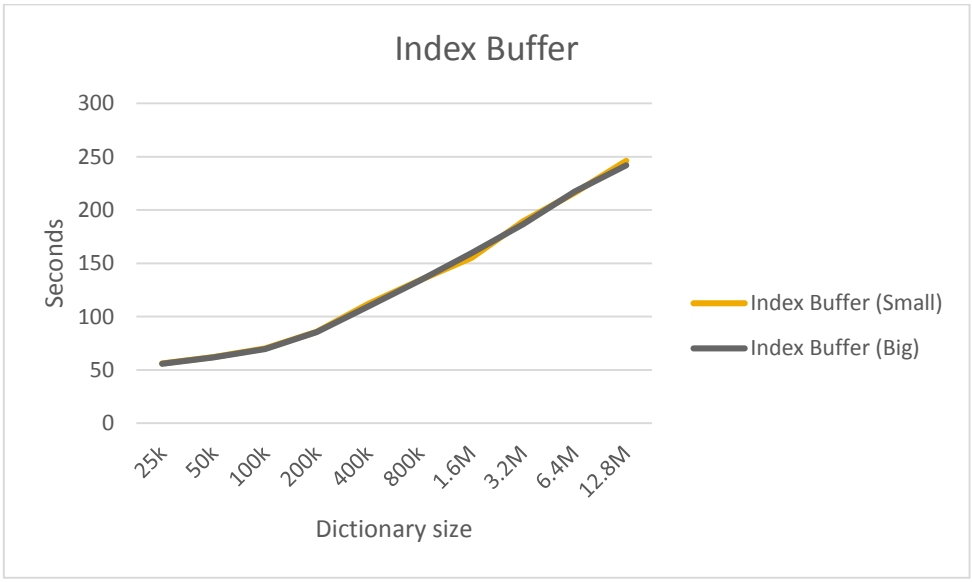


Figure 25: Insertion times respect data distribution in Index Buffer

7.2.3. Standard Buffer

The next approach to be analyzed is the Standard Buffer. This buffer first stores multiple records without compression and then saves them all in the differential store. The Figure 26 shows the insertion times for the different data distribution respect to the data that was already inserted. It shows something similar than in the previous cases, that the worst case is at the beginning when is building the dictionary, and then it becomes more or less constant. Another aspect is that this type of buffer suffers of more fluctuations in the insertion times than the previous cases. About the Figure 27, it shows how the size of the buffer affects to the insertion times. The details of the intermediate buffer sizes can be found in the index of tables, at the end of the document. But analyzing the smallest size of 100,000 values and the biggest of 12,800,000

values, it can be seen that the more repeated values the best behaves the biggest buffer. But when the entropy of the data is higher, the big buffers have worst insertion times. The explanation of this is that the higher the entropy is the bigger will be the dictionary and hence the tree, so the addresses of the values in the buffer will not be longer in the cache. This added to the time increase because of the size of the tree, creates the curve of growing time for the bigger buffers.

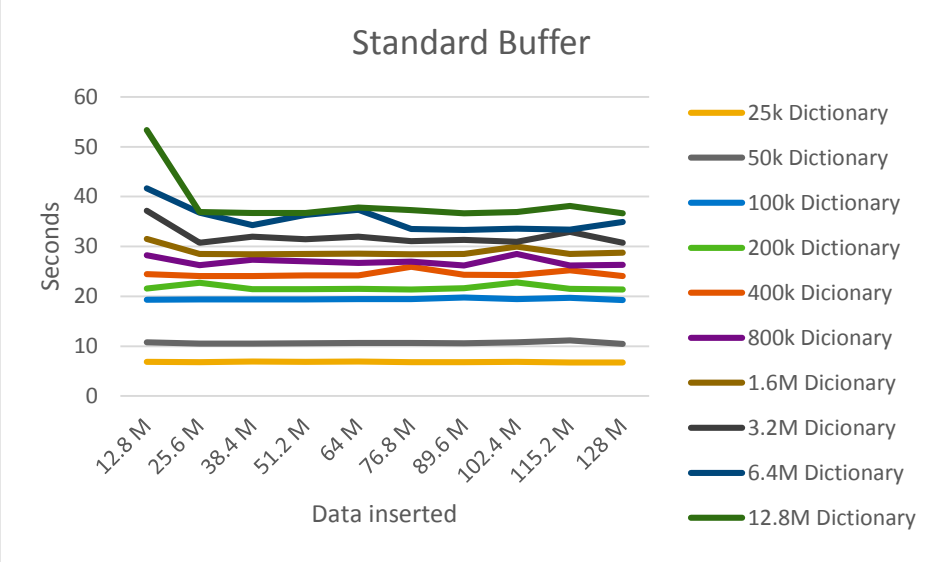


Figure 26: Insertion times respect data already inserted in Standard Buffer

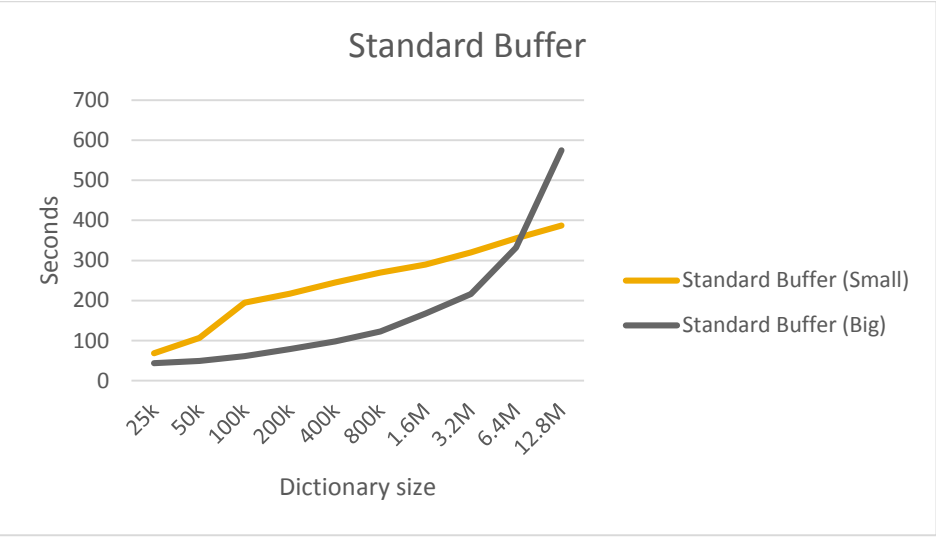


Figure 27: Insertion times respect data distribution in Standard Buffer

7.2.4. Sort Buffer

The following strategy for the buffer is the most complex of the prototype. This one, as it was exposed in the design chapter, is based on save accesses to the dictionary of the differential store. The Figure 28 shows, as in the previous cases, the insertion times for the different data distribution respect data that was already inserted. Since all the data that is going to be inserted

in the system will be sorted within the buffer, there are less fluctuations because those are probably produced because of the different accesses to the tree. Respect the Figure 29, it shows the benefits and tradeoffs of increasing the size of the buffer. As it can be seen in the buffer, for the lowest entropy the bigger the buffer is the worst time produces. This is because it takes more time to sort all the values in the buffer than really look up for the value in the tree. But this is only for the slowest entropy, for all the values from there it is worth the time spent sorting the data in exchange to save looks up in the dictionary. And for the highest entropy, even when every value within the buffer is different, it also improves the insertion. This is because as explained in [30] and [31], the insertion of sorted data in a tree is more effective than the unsorted insertion of exactly the same data.

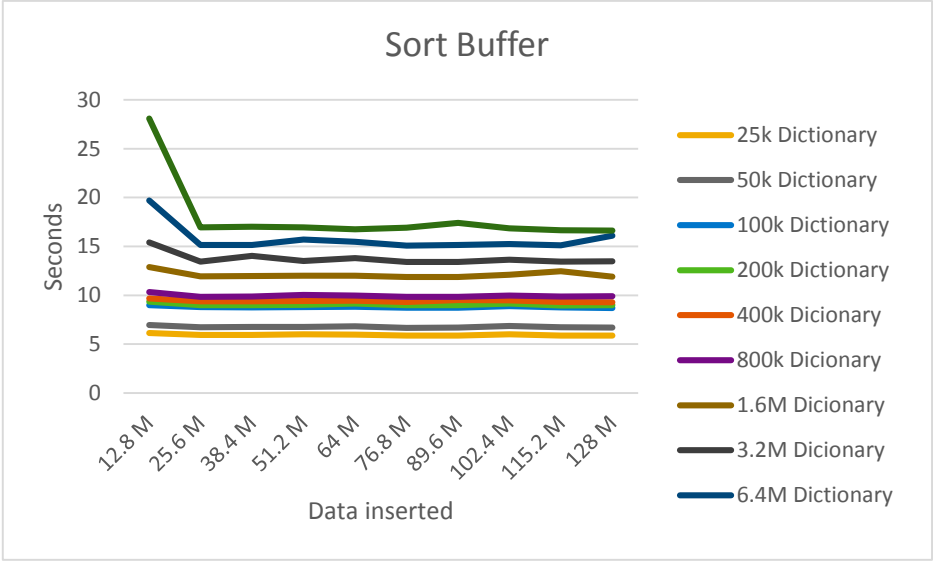


Figure 28: Insertion times respect data already inserted in Sort Buffer

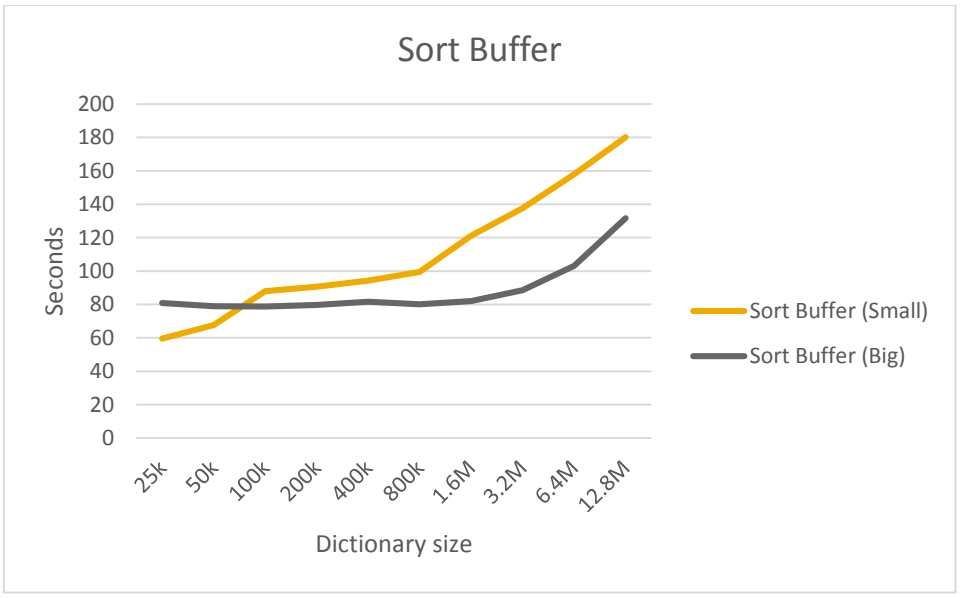


Figure 29: Insertion times respect data distribution in Sort Buffer

7.2.5. Map Buffer

The last approach that was proved in the system was the addition of an unsorted structure in the merge phase to check if it could improve the merge times. The Figure 30 indicates, as in the other approaches, the insertion times for the different data distribution respect data that was already inserted. And the Figure 31 compare the smallest size of the buffer of 100,000 values and the biggest size of 12,800,000 values. Something interesting is that the insertion times in both graphics are very similar to the ones of the standard buffer. This can lead to the conclusion that the addition of the unsorted structure does not improve the performance in the merge. The explanation is that the look up in the structure of the buffer in order to find the value takes as long as it takes to find the value in the differential store. So, basically the performance of both approaches is very similar.

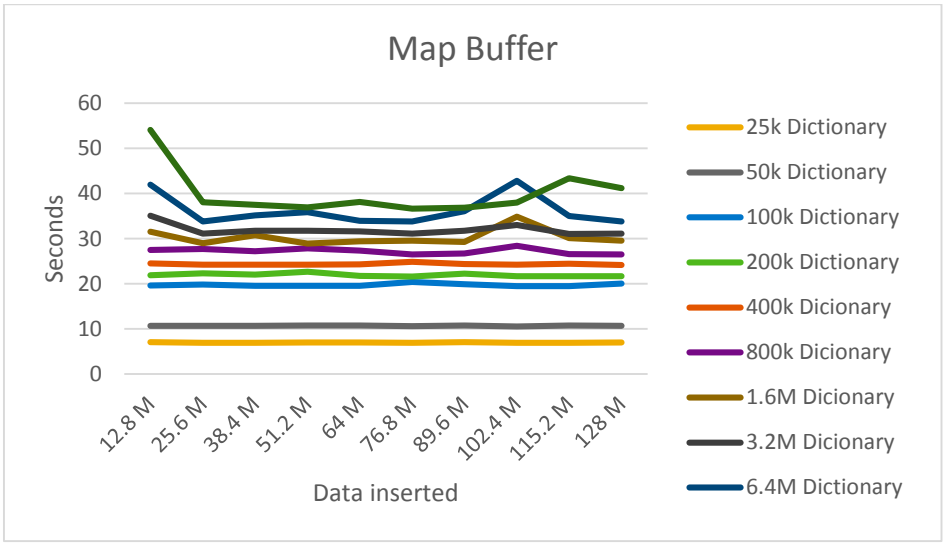


Figure 30: Insertion times respect data already inserted in Map Buffer

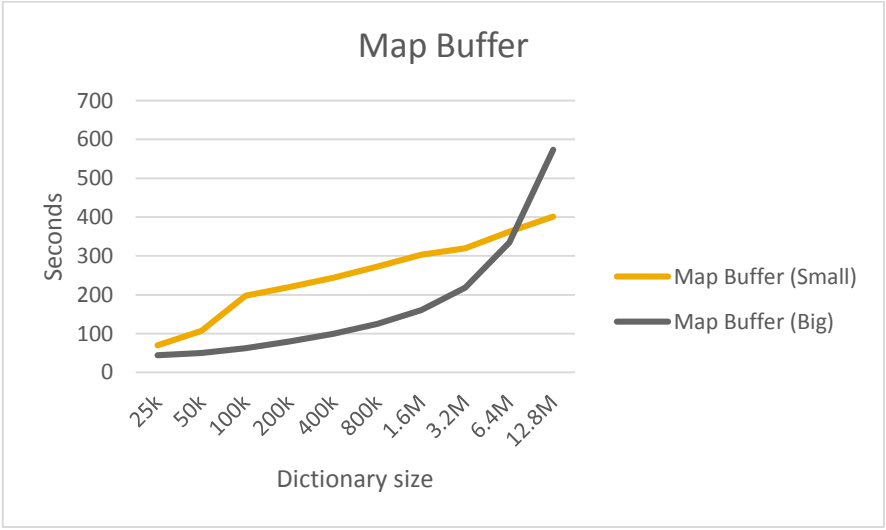


Figure 31: Insertion times respect data distribution in Map Buffer

7.2.6. Insertion comparison

Once each approach is studied separately and it has been understood more deeply, in this section, all the different strategies for the prototype are going to be compare to each other. The goal is to find the best strategy concerning to the insertion/merge process. The results of the comparison can be found in the Figure 32 and the Figure 33. The first one compares the different strategies using the smallest sizes of the buffers. The second, compares the different approaches for the prototype using the biggest buffers.

First of all, it stands out that the performance of disable the buffer or using the index buffer, no matters the size of the buffer, is mostly the same.

After, it can be found a second group that involves the map buffer and the standard buffer, for reasons explained in the above sections the performance is very similar to each other. But as it can be appreciated in the figures of comparison, the performance for the smallest sizes is similar for the smaller entropy of data and it becomes worse than the use of no buffer at all. With the increase of the size this effect is minimized but in general it only reaches a performance similar to the use of no buffer at all and only for the lowest entropy of data.

And the last strategy, the sort buffer, it has a different behavior than the others. For the smaller entropy of data it can be appreciated that its performance it is clearly worse than any of the other. In the other hand, for the higher entropy, from 400,000 different values, it can be seen that the sort of data before merge it improves the insertion time respect the other strategies. Even when every value is different, and hence the worst case for every approach, the sort buffer makes better times than the other buffers.

In conclusion, for data with low entropy the most recommendable is the use of no buffer at all or the index buffer. While when the data has a high entropy, the most recommendable approach is the use of the sort buffer.

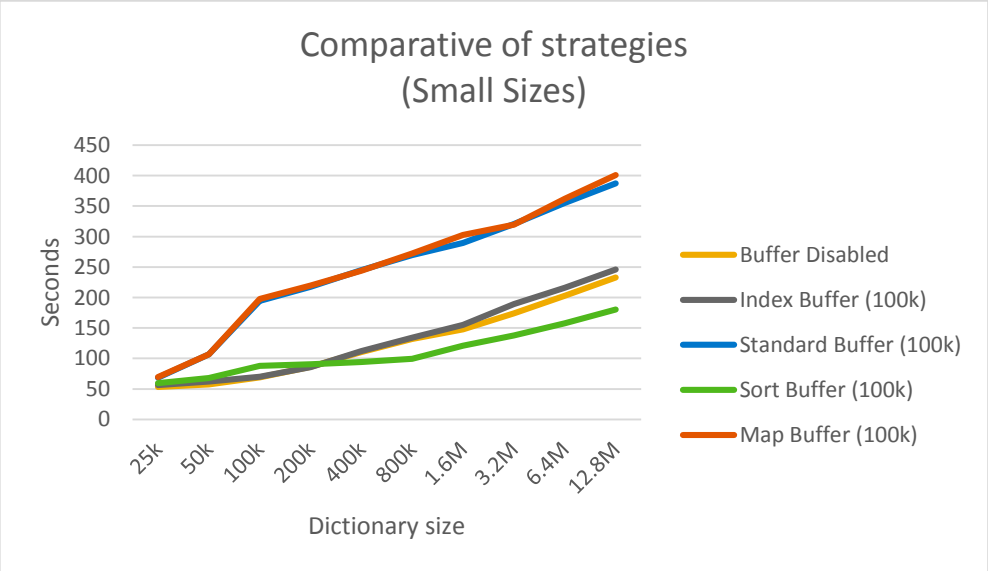


Figure 32: Comparative of strategies using the smallest buffer sizes in insertion

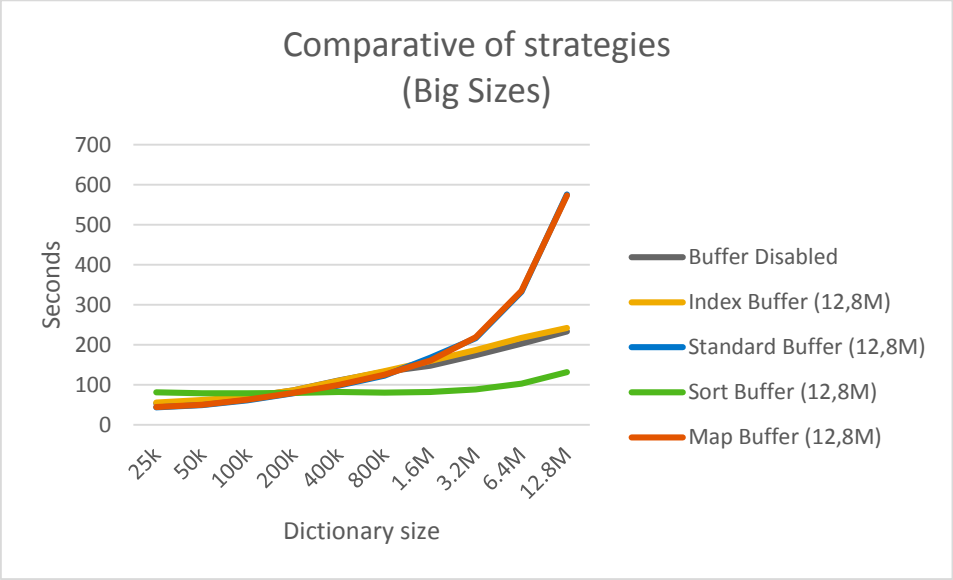


Figure 33: Comparative of strategies using the smallest buffer sizes in insertion

7.3.Query performance

The second aspect it has to be analyzed is the read performance. The read operations defined in the system were explained in the design chapter, in the use cases. Although this aspect is a bit more secondary in the evaluation, because the main aspect to be optimize is the insertion, it cannot be forgotten the impact in the read operations.

There are two decisive parameters in these test. First of all, the number of values inserted in the system on which the tests are going to be run. And second, the number of read operations that are going to be run over a specific set of data.

Regarding to the first aspect, each time the system will run the tests over the double of data than in the previous time. Starting from 100,000 values, it will double the number of data until it reaches 12,800,000 values. This pretends to analyze the impact that has the number of values in the performance of each read operation.

About the second characteristic, it has been specified a fixed number of operations for each type. The main reason of this is because, always that the size of the result is approximately the same, the number of operations that are executed for each query is always the same. For example, for a *select* it will, transform in case it is necessary the value to the position in the dictionary and, check every position in the index. If the test runs the same select multiple times, it will only repeat the same operations each time. And although the data to be used for the queries is random, it is always generated using the same seed. This makes that each time the same data is used. Knowing that, it has been designated a number of operations that is big enough to be representative, but not too large to make the test unviable. After multiple measures, it was chosen 100 times for each type of operation as a desirable value.

In the following sections each of the three different operations is going to be analyzed separately. For each one, the performance between the different strategies is going to be

studied. The Map buffer is not showed in the graphics because the data structure until the merge operation is the same than in the Standard buffer. The assumptions are based in the data that can be found in the index of tables. In order to facilitate the read and comprehension of the different evaluations, only resuming graphics are going to be showed. The graphic are the total times of executing 100 operations every time the system runs a block of insertions.

7.3.1. Select comparison

Looking at the performance of the different approaches in the select operation, it stands out the advantages of the dictionary compression. In order to run through all the values stored and compared them to the one the system is looking for, the dictionary compression improves, in orders of magnitude, the performance. The two techniques that takes advantage of the dictionary compression, have significant lower times executions. While the approaches that have not compression have a performance approximated of 5 times higher.

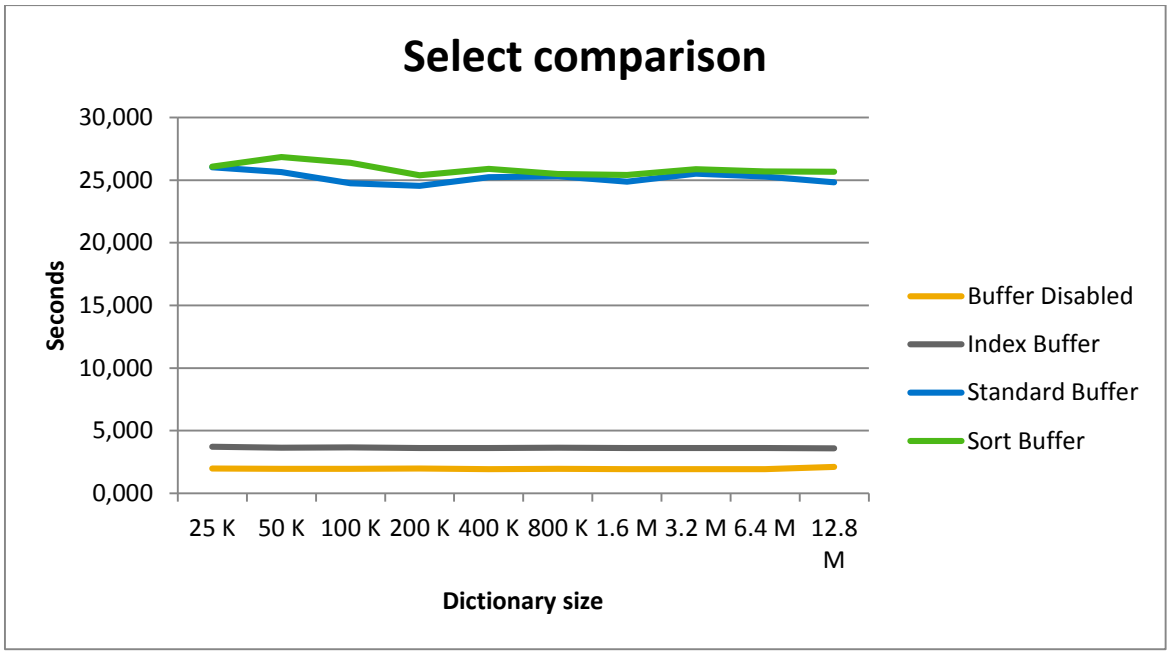


Figure 34: Comparison of select performance

7.3.2. Scan comparison

Regarding to return whole columns of attributes, the performance of the different strategies is more similar to each other than in the previous case. Any way it can still be found that the system executes it more efficiently when is not using the buffer. This performance decreases with the complexity used, as happens in the select operation. The index buffer has the performance closer to the use of the buffer disabled, while the standard buffer and the sort buffer has the worst performance.

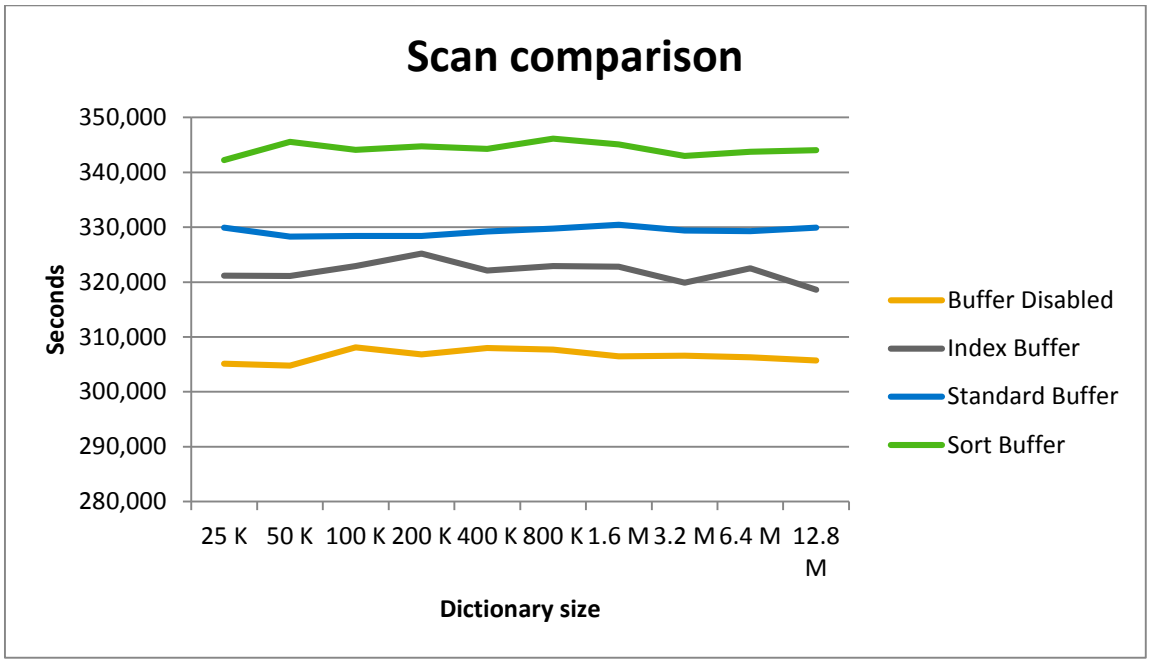


Figure 35: Comparison of scan performance

7.3.3. Range comparison

The range operation is the only one where it can be appreciated the improvement of the more complex buffers. The use of the buffer disabled or the index buffer produce worst times, and they increase with the size of the dictionary. This is because of the size of the tree, the more different values are in the system, the more values it has to be considered to be in the range. But in the other hand, when the system uses buffers without dictionary compression, this improves the performance, because it only needs to compare the values. It does not need to decompress the value using the dictionary.

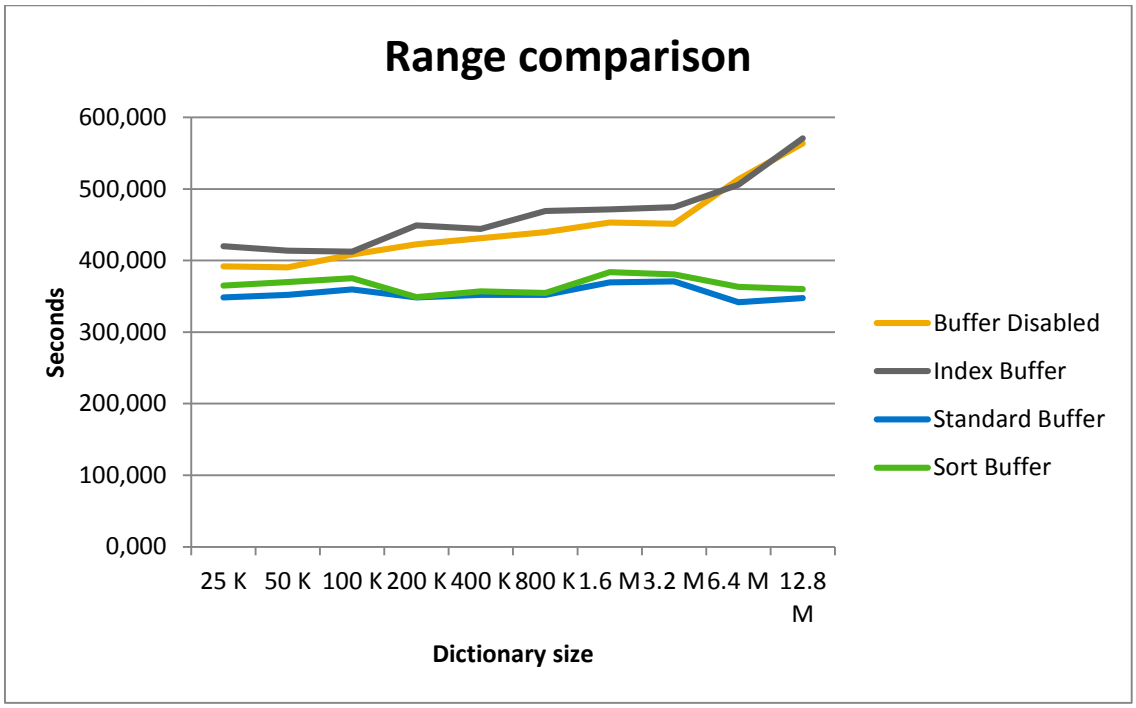


Figure 36: Comparison of range performance

8. Conclusion

One of the definitive aspects that has a big repercussion in the performance analyzed in the evaluation chapter is the data distribution, in other words the size of the dictionary. Since it is obvious depending of the data distribution, there is a better approach or other, a new strategy was created. In the design and implementation chapter, it was explained that through the use of polymorphism the system used all the different approaches without knowing exactly which one was executing. This advantage in the design of the system was taking in advantage and use to create a final approach. Accessing to the size of the index and the size of the dictionary shows the data distribution that was used in that attribute of the table so far. Knowing that, it can be predicted the nature of the future incoming data. That allows that after a merge operation, the system changes the approach used for that attribute until that moment. This last strategy took the name of hybrid buffer.

A combination of the index buffer and the sort buffer was selected. The evaluation chapter explains the advantages of using an index buffer for small dictionaries while for bigger dictionaries the sort buffer performs better. The ratio of change was set up based in the following equations:

	Dictionary of 200,000	Dictionary of 400,000
Index Buffer	85.801	110.61
Sort Buffer	90.577	94.245

Knowing the point where there performance of the Sort Buffer and the Index Buffer crosses, the equation of the line that represents the performance for each buffer is:

$$\left(\frac{y-y_1}{y_2-y_1}\right)=\left(\frac{x-x_1}{x_2-x_1}\right)$$

$$Index\ Buffer: \left(\frac{y-58.801}{100.61-85.801}\right)=\left(\frac{x-200}{400-200}\right)$$
$$Index\ Buffer:\ -24.809x+200y=12198.4$$

$$Sort\ Buffer: \left(\frac{y-90.577}{94.245-90.577}\right)=\left(\frac{x-200}{400-200}\right)$$
$$Sort\ Buffer:\ -3.668x+200y=18849$$

$$Equation\ system: \begin{cases} -24.809x+200y=12198.4 \\ -3.668x+200y=18849 \end{cases}$$

$$x=314.5830\ Dictionary\ Size\ where\ the\ performance\ crosses$$

$$\frac{314.5830}{12{,}800}=40.6887\ ratio\ index/dictionary$$

The insertion performance of this approach is a combination of the index buffer and the sort buffer. As it can be appreciated in the figure 37, for dictionaries smaller than 400,000 values uses the index buffer. And for dictionaries bigger, uses the sort buffer.

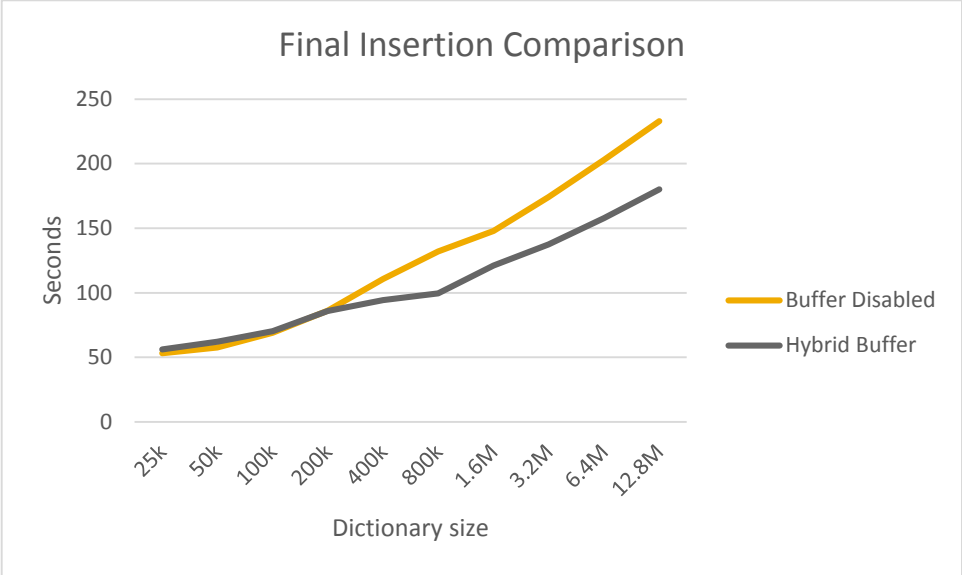


Figure 37: Insertion performance of Hybrid Buffer

Regarding to the query performance, the performance here is more complex to analyze. It looks worse than the use of no buffer at all. The select performance is clearly worse, as it shows the figure 38. The scan performance is closer to the use of no buffer, but still is worse, as it shows the figure 39. And the last, the range performance is slightly better, but not considerably, as it shows the figure 40.

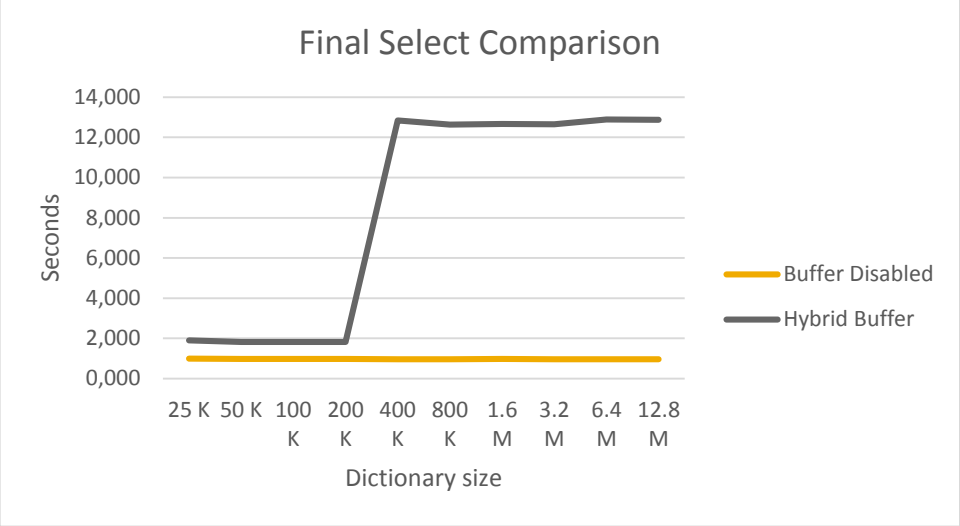


Figure 38: Select performance of Hybrid Buffer

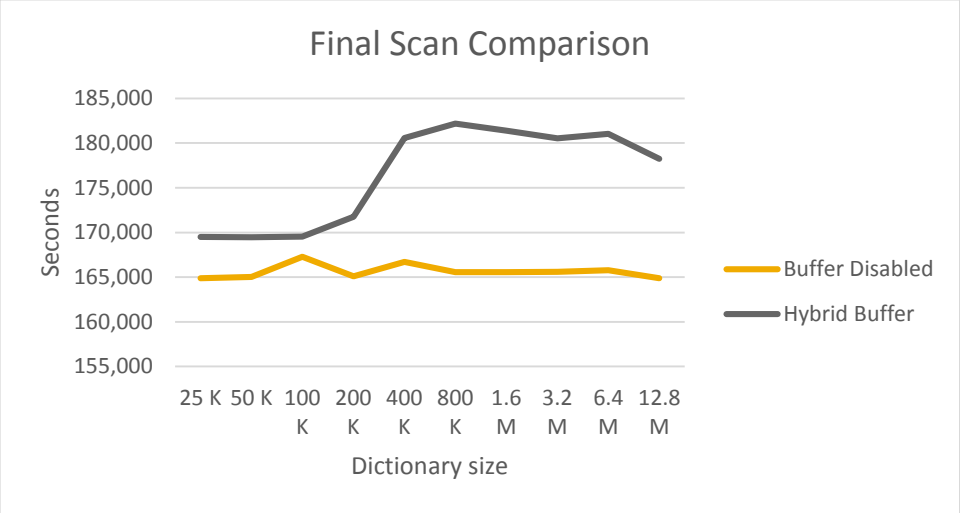


Figure 39: Scan performance of Hybrid Buffer

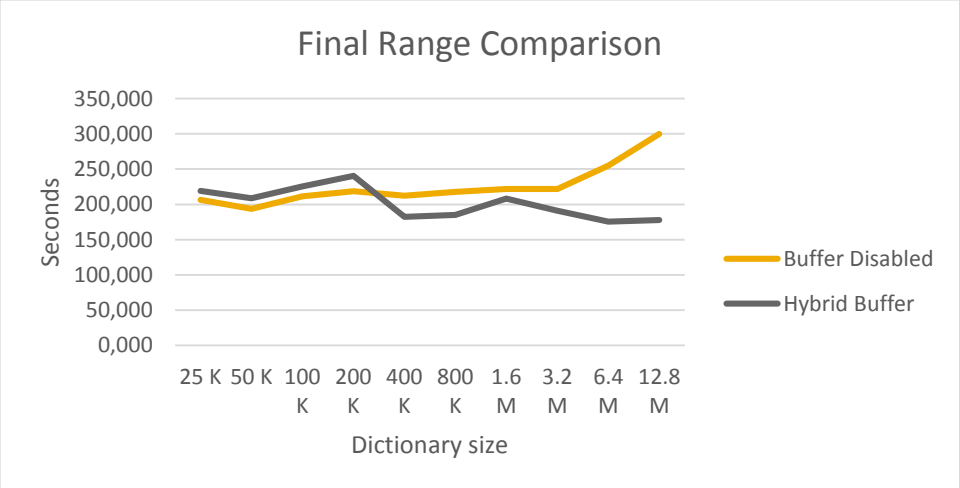


Figure 40: Range performance of Hybrid Buffer

The main goal of the project was to improve the insertion performance, trying to reduce the impacts of the query performance as much as possible. And it is consider that this was achieved. The insertion is obviously improved, but at the expense of the query performance. Anyway the prototype is based in the SAP HANA database, which has more complex structures and strategies to run the read operations that those it have been used here.

8.1.Future work

Based in [9] and [15], it would be very interesting to develop the merge asynchronous. This would allow to run the merge operation in the background while allowing to insert and query in the new buffer. The best approach for the problem would be to use a combination of the Standard Buffer and the Sort buffer for the Hybrid buffer. This is because for the smaller dictionary sizes the performance of the Index buffer and the Standard buffer it is very similar. But, as it can be found in the results, the main load of the system using the Standard buffer happens in the merge, as it happens in the Sort Buffer. While for the Index Buffer, it happens in the insertion of the buffer. If the load of the system is moved to the merge operation, and that happens in the background it is believed the performance would improve considerably.

Another remarkable aspect to study could be the development of an algorithm that detects when the system is getting a heavy load of read operation and then activates the merge operation, instead of using a fixed size. For example, based in the internet protocols that manages the sizes of the queues dynamically. This would improve even further the performance of the system.

11. Legal Framework

Since the project is focus to my home university which is placed in Spain, and also because I do not have any kind of formation in the legal framework of Germany, this section is going to explain the regulation that affects the project in Spain.

Since this application is meant to storage personal details in the form of a database, is affected to the Organic Law of data protection. The law 15/1999 of Personal Data Protection is in charge to protect and ensure the honor, intimacy and privacy of the subject and this family.

It also specifies the obligations that have to be taken for those who are responsible of the storage data. The responsible, public or private agencies, have to guarantee the protection of all the data that is considered personal, such as passport number, national document number, address, telephone numbers, etc…

Although the supposition of the project is a data base, in the reality is not a real product and it was not using personal or confidential data in any moment. This way the people involved in the development did not have to take any prevention measure.

9. Budget planning

The budget of the project takes in consideration multiple aspects. First of all the hardware used during the development of the project.

Item	Cost per unit	Number of units	Total
Workstation z600	1.600,00 €	1	1.600,00 €
ThinkPad W530 Laptop	2.155,00 €	1	2.155,00 €
		Total	3.755,00 €

Also the software and their respective licenses. In the case of the Windows 7 and the Microsoft Office 2010, both were installed in the main workstation and the laptop.

Item	Cost per unit	Number of units	Total
Visual Studio	500,00 €	1	500,00 €
Intel Vtune Amplifier XE	899,00 €	1	899,00 €
Windows 7	164,00 €	1	164,00 €
Microsoft Office 2010	199,00 €	1	199,00 €
		Total	1.762,00 €

The project had a duration of 6 months, so it is necessary to adjust the budget for those items to the time they have been in used and have an estimated life of 30 months. So the amortization cost are the following.

Item	Total Cost	Cost per month	Amortization cost
Hardware	1.762,00 €	58,73 €	352,40 €
Software	3.755,00 €	125,17 €	751,00 €
		Total	1.103,40 €

Another remarkable aspect is the cost of the people involved in the development of the project.

Item	Cost per hour	Number of hours	Total
Intern	5,00 €	960	4.800,00 €

Since the whole development was done in a different country of my home university, it is considered important the cost of the transportation.

Item	Cost per unit	Number of units	Total
Airplane tickets	120,00 €	2	240,00 €

The result of joining all the previous costs shows the total cost of the project.

Item	Total
Hardware	352,40 €
Software	751,00 €
Personal	4.800,00 €
Transport	240,00 €
Total	6.143,40 €

10. References

1. Edgard Codd. A relational model of data for large shared data banks. Comunications of the ACM, volume 26 Issue 1, Jan. 1983.
2. GC33-0754-00
3. G. E. Moore. Cramming More Components onto Integrated Circuits. Electronics, 38 pp. 114-117, Apr. 1965.
4. Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, Stan Zdonik. C-store: A column-oriented DBMS. 31th VLDB Conference, 2005.
5. Peter Boncz, Stefan Manegold, Martin L. Kersten. Database architecture optimized for the new bottleneck: memory access. 25th VLDB Conference, 1999.
6. Jens Krueger, Martin Grund, Christian Tinnefeld, Hasso Plattner, Alexander Zeier, and Frankz Faerber. Optimizing write performance for read optimized databases. DASFAA'10 Proceedings of the 15th international conference on Database Systems for Advanced Applications - Volume Part II, pages 291-305.
7. D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In Proceedings of the 2006 ACM SIGMOD international conference on Management of data, pages 671–682. ACM, 2006
8. Jun Rao, Kenneth A. Ross. Making B+ -Trees Cache Conscious in Main Memory. MOD, 2000.
9. Florian Hübner, Joos-Hendrik Boese, Jens Krüger, Frank Renkes, Cafer Tosun, Alexander Zeier, Hasso Plattner: A Cost-Aware Strategy for Merging Differential Stores in Column-Oriented In-Memory DBMS, BIRTE – in conjunction with VLDB'11, 2011.
10. Jens Krueger, Changkyu Kim, Martin Grund, Nadathur Satish, David Schwalb, Jatin Chhugani, Hasso Plattner, Pradeep Dubey, Alexander Zeier. Fast Updates on Read-Optimized Databases Using Multi-Core CPUs. VLDB Conference, 2011.
11. J. Krueger, C. Tinnefeld, M. Grund, A. Zeier, and H. Plattner. A Case for Online Mixed Workload Processing. In 3rd International Workshop on Testing Database System, 2010.
12. Anja Bog, Kai Sachs, Alexander Zeier, and Hasso Plattner. Normalization in a Mixed OLTP and OLAP Workload Scenario. In TPC TC, 2011.
13. Allison L. Holloway, David J. DeWitt. Read-Optimized Databases, In Depth. In VLD, 2008.
14. Christian Lemke, Kai-Uwe Sattler, Franz Faerber, and Alexander Zeier. Speeding Up Queries in Column Stores. DaWaK, pp. 117-129, 2010.
15. Jens Krueger, Martin Grund, Christian Tinnefeld, Hasso Plattner, Alexander Zeier, and Franz Faerber. Optimizing Write Performance for Read Optimized Databases. DASFAA, pages 291-305, 2010.
16. Robert W. Taylor, Randall L. Frank. CODASYL Data-Base Management Systems. Computer Surveys, Vol. 8, No 1, March 1976.
17. Javier Quiroz. El modelo relacional de bases de datos. Boletin de Politica Informatica, no 6, 2003.
18. Michael Stonebraker. The INGRES papers: Anatomy of a relational database system. Addison-Wesley, 1986.
19. M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and

V. Watson. System R: Relational Approach to Database Management. ACM Transactions on Database Systems, Vol. 1, No. 2. June 1976.

20. Donald D. Chamberlin , Morton M. Astrahan , Michael W. Blasgen , James N. Gray , W. Frank King , Bruce G. Lindsay , Raymond Lorie , James W. Mehl , Thomas G. Price , Franco Putzolu , Patricia Griffiths Selinger , Mario Schkolnick , Donald R. Slutz , Irving L. Traiger , Bradford W. Wade , Robert A. Yost, A history and evaluation of System R, Communications of the ACM, v.24 n.10, p.632-646, Oct. 1981.
21. D. J. Abadi. Query Execution in Column-Oriented Database Systems. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, Feb 2008
22. W.H. Inmon. Building the Data Warehouse. John Wiley, 1992.
23. Surajit Chaudhuri, Umeshwar Dayal. An Overview of Data Warehousing and OLAP Technology. ACM Sigmod Record, March 1997.
24. W. H. Immon. Operational and Informational Reporting: Information Management: Charting the Course. DM Review Magazine, July 2000.
25. Jan Schaffner, Anja Bog, Jens Krüger, and Alexander Zeier. A Hybrid Row-Column OLTP Database Architecture for Operational Reporting.
26. G. P. Copeland, and S. Khoshafian. A Decomposition Storage Model. SIGMOD conference, pp. 268-279, 1985.
27. Franz Färber, Norman May, Wolfgang Lehner, Philipp Grosse, Ingo Müller, Hannes Rauhe, and Jonathan Dees. The SAP HANA Database – An Architecture Overview. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 2012.
28. Hasso Plattner. A common Database Approach for OLTP and OLAP using an In-Memory Column Database. SIGMOD, June 29-July 2, 2009.
29. Vishal Sikka, Frankz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, Christof Bornhövd. Efficient Transaction Processing in SAP HANA Database – The End of a Column Store Myth. SIGMOD, May 20-24, 2012.
30. Jochen van den Bercken, and Bernhard Seeger. An evaluation of Generic Bulk Loading Techniques. VLDB Conference, 2001.
31. Riku Saikkonen, and Eljas Soisalon-Soininen. Bulk-Insertion Sort: Towards Composite Measures of Presortedness. J. Vahrenhold (Ed.): SEA 2009, LNCS 5526, pp. 269–280, 2009.

11. Index of tables

11.1. Insert and merge results

25,000 values in dictionary

No Buffer

1 Column with dictionary of size 25K - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	0 K	5.321	5.289	5.287	5.321	5.318	5.394	5.335	5.409	5.385	5.232	53.291

1 Column with dictionary of size 25K - Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	0 K	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

1 Column with dictionary of size 25K - Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	0 K	5.321	5.289	5.287	5.321	5.318	5.394	5.335	5.409	5.385	5.232	53.291

Index Buffer

1 Column with dictionary of size 25K - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	5.646	5.551	5.521	5.490	5.512	5.574	5.507	5.505	5.518	5.533	55.357
	200 K	5.625	5.496	5.559	5.455	5.486	5.476	5.468	5.431	5.474	5.472	54.942
	400 K	5.627	5.512	5.511	5.531	5.519	5.503	5.526	5.508	5.478	5.465	55.180
	800 K	5.568	5.521	5.625	5.481	5.540	5.511	5.505	5.522	5.491	5.475	55.239
	1.6 M	5.509	5.495	5.530	5.491	5.494	5.532	5.515	5.518	5.558	5.480	55.122
	3.2 M	5.571	5.505	5.488	5.524	5.482	5.474	5.664	5.530	5.571	5.458	55.267
	6.4 M	5.571	5.518	5.609	5.540	5.516	5.487	5.498	5.542	5.547	5.525	55.353
	12.8 M	5.538	5.475	5.551	5.524	5.492	5.551	5.516	5.476	5.536	5.502	55.161

1 Column with dictionary of size 25K - Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	0.108	0.070	0.075	0.108	0.138	0.038	0.199	0.029	0.035	0.032	0.832
	200 K	0.088	0.097	0.091	0.022	0.124	0.028	0.175	0.033	0.022	0.228	0.908
	400 K	0.081	0.093	0.083	0.118	0.031	0.161	0.030	0.030	0.218	0.027	0.872
	800 K	0.104	0.060	0.079	0.103	0.146	0.028	0.026	0.198	0.030	0.027	0.801
	1.6 M	0.078	0.103	0.097	0.030	0.137	0.029	0.185	0.030	0.028	0.261	0.978
	3.2 M	0.065	0.089	0.088	0.125	0.029	0.172	0.034	0.030	0.250	0.028	0.910
	6.4 M	0.041	0.090	0.093	0.100	0.030	0.163	0.030	0.225	0.030	0.030	0.832
	12.8 M	0.029	0.055	0.080	0.105	0.129	0.030	0.168	0.030	0.030	0.029	0.685

1 Column with dictionary of size 25K - Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	5.754	5.621	5.596	5.598	5.650	5.612	5.706	5.534	5.553	5.565	56.189
	200 K	5.713	5.593	5.650	5.477	5.610	5.504	5.643	5.464	5.496	5.700	55.850
	400 K	5.708	5.605	5.594	5.649	5.550	5.664	5.556	5.538	5.696	5.492	56.052
	800 K	5.672	5.581	5.704	5.584	5.686	5.539	5.531	5.720	5.521	5.502	56.040
	1.6 M	5.587	5.598	5.627	5.521	5.631	5.561	5.700	5.548	5.586	5.741	56.100
	3.2 M	5.636	5.594	5.576	5.649	5.511	5.646	5.698	5.560	5.821	5.486	56.177
	6.4 M	5.612	5.608	5.702	5.640	5.546	5.650	5.528	5.767	5.577	5.555	56.185
	12.8 M	5.567	5.530	5.631	5.629	5.621	5.581	5.684	5.506	5.566	5.531	55.846

Standard Buffer

1 Column with dictionary of size 25K - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	0.568	0.561	0.584	0.562	0.587	0.578	0.584	0.525	0.547	0.536	5.632
	200 K	0.583	0.567	0.555	0.535	0.576	0.552	0.555	0.541	0.584	0.558	5.606
	400 K	0.577	0.543	0.559	0.550	0.559	0.576	0.545	0.567	0.550	0.563	5.589
	800 K	0.578	0.531	0.562	0.552	0.537	0.562	0.600	0.574	0.588	0.562	5.646
	1.6 M	0.589	0.595	0.552	0.560	0.554	0.571	0.556	0.596	0.539	0.577	5.689
	3.2 M	0.587	0.564	0.568	0.582	0.546	0.575	0.557	0.569	0.538	0.577	5.663
	6.4 M	0.634	0.579	0.585	0.573	0.530	0.578	0.545	0.557	0.550	0.567	5.698
	12.8 M	0.680	0.581	0.569	0.563	0.561	0.545	0.565	0.612	0.591	0.586	5.853

1 Column with dictionary of size 25K - Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	6.282	6.239	6.349	6.289	6.330	6.189	6.192	6.357	6.203	6.190	62.620
	200 K	5.089	5.051	5.055	5.086	5.130	5.069	5.042	5.230	5.018	5.027	50.797
	400 K	4.444	4.401	4.418	4.444	4.480	4.375	4.366	4.554	4.378	4.367	44.227
	800 K	4.110	4.066	4.074	4.103	4.150	4.063	4.025	4.215	4.028	4.025	40.859
	1.6 M	3.933	3.895	3.915	3.932	3.981	3.853	3.853	4.031	3.853	3.866	39.112
	3.2 M	3.883	3.854	3.866	3.896	3.991	3.819	3.812	4.048	3.834	3.829	38.832
	6.4 M	3.834	3.766	3.783	3.812	3.855	3.734	3.794	3.983	3.760	3.737	38.058
	12.8 M	3.810	3.745	3.776	3.796	3.822	3.731	3.734	3.900	3.731	3.720	37.765

1 Column with dictionary of size 25K - Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	6.850	6.800	6.933	6.851	6.917	6.767	6.776	6.882	6.750	6.726	68.252
	200 K	5.672	5.618	5.610	5.621	5.706	5.621	5.597	5.771	5.602	5.585	56.403
	400 K	5.021	4.944	4.977	4.994	5.039	4.951	4.911	5.121	4.928	4.930	49.816
	800 K	4.688	4.597	4.636	4.655	4.687	4.625	4.625	4.789	4.616	4.587	46.505
	1.6 M	4.522	4.490	4.467	4.492	4.535	4.424	4.409	4.627	4.392	4.443	44.801
	3.2 M	4.470	4.418	4.434	4.478	4.537	4.394	4.369	4.617	4.372	4.406	44.495
	6.4 M	4.468	4.345	4.368	4.385	4.385	4.312	4.339	4.540	4.310	4.304	43.756
	12.8 M	4.490	4.326	4.345	4.359	4.383	4.276	4.299	4.512	4.322	4.306	43.618

Sort Buffer

1 Column with dictionary of size 25K - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	0.829	0.817	0.822	0.828	0.782	0.819	0.807	0.780	0.802	0.783	8.069
	200 K	0.826	0.803	0.791	0.822	0.824	0.799	0.760	0.820	0.828	0.788	8.061
	400 K	0.826	0.797	0.820	0.797	0.826	0.810	0.834	0.806	0.811	0.838	8.165
	800 K	0.820	0.815	0.792	0.808	0.808	0.807	0.827	0.827	0.850	0.816	8.170
	1.6 M	0.849	0.818	0.786	0.847	0.807	0.815	0.776	0.788	0.857	0.831	8.174
	3.2 M	0.833	0.825	0.806	0.827	0.831	0.815	0.801	0.839	0.900	0.842	8.319
	6.4 M	0.894	0.812	0.818	0.794	0.842	0.813	0.779	0.809	0.823	0.792	8.176
	12.8 M	0.947	0.824	0.828	0.794	0.807	0.818	0.801	0.822	0.782	0.807	8.230

1 Column with dictionary of size 25K - Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	5.323	5.114	5.126	5.196	5.183	5.062	5.074	5.239	5.071	5.101	51.489
	200 K	5.368	5.166	5.236	5.207	5.295	5.139	5.136	5.339	5.165	5.160	52.211
	400 K	5.759	5.558	5.589	5.602	5.636	5.517	5.522	5.708	6.239	5.539	56.669
	800 K	5.933	5.611	5.624	5.653	5.676	5.576	5.565	5.765	5.570	5.563	56.536
	1.6 M	6.458	6.077	6.081	6.108	6.134	6.032	6.204	6.193	6.065	6.072	61.424
	3.2 M	6.910	6.152	6.156	6.600	6.234	6.115	6.528	6.278	7.581	6.137	64.691
	6.4 M	9.039	6.662	6.723	6.710	6.721	6.619	6.621	7.242	7.117	6.610	70.064
	12.8 M	9.829	7.144	6.799	7.038	6.856	6.741	7.816	6.932	6.748	6.760	72.663

1 Column with dictionary of size 25K - Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	6.152	5.931	5.948	6.024	5.965	5.881	5.881	6.019	5.873	5.884	59.558
	200 K	6.194	5.969	6.027	6.029	6.119	5.938	5.896	6.159	5.993	5.948	60.272
	400 K	6.585	6.355	6.409	6.399	6.462	6.327	6.356	6.514	7.050	6.377	64.834
	800 K	6.753	6.426	6.416	6.461	6.484	6.383	6.392	6.592	6.420	6.379	64.706
	1.6 M	7.307	6.895	6.867	6.955	6.941	6.847	6.980	6.981	6.922	6.903	69.598
	3.2 M	7.743	6.977	6.962	7.427	7.065	6.930	7.329	7.117	8.481	6.979	73.010
	6.4 M	9.933	7.474	7.541	7.504	7.563	7.432	7.400	8.051	7.940	7.402	78.240
	12.8 M	10.776	7.968	7.627	7.832	7.663	7.559	8.617	7.754	7.530	7.567	80.893

Map Buffer

1 Column with dictionary of size 25K - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	0.559	0.554	0.555	0.532	0.566	0.564	0.539	0.561	0.561	0.582	5.573
	200 K	0.542	0.555	0.536	0.588	0.575	0.571	0.564	0.545	0.554	0.551	5.581
	400 K	0.559	0.561	0.586	0.559	0.574	0.572	0.545	0.554	0.574	0.530	5.614
	800 K	0.566	0.549	0.565	0.542	0.565	0.560	0.547	0.560	0.553	0.603	5.610
	1.6 M	0.586	0.560	0.576	0.585	0.552	0.588	0.556	0.571	0.553	0.553	5.680
	3.2 M	0.582	0.571	0.587	0.572	0.566	0.581	0.561	0.587	0.593	0.577	5.777
	6.4 M	0.659	0.559	0.565	0.565	0.567	0.574	0.582	0.553	0.558	0.586	5.768
	12.8 M	0.637	0.567	0.552	0.550	0.562	0.577	0.567	0.550	0.535	0.550	5.647

1 Column with dictionary of size 25K - Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	6.478	6.368	6.379	6.439	6.463	6.342	6.537	6.363	6.393	6.411	64.173
	200 K	5.157	5.161	5.156	5.086	5.180	5.114	5.240	5.088	5.090	5.302	51.574
	400 K	4.537	4.534	4.544	4.560	4.464	4.615	4.470	4.476	4.684	4.498	45.382
	800 K	4.228	4.192	4.234	4.274	4.283	4.185	4.161	4.355	4.135	4.207	42.254
	1.6 M	4.080	4.113	4.093	4.028	4.118	4.123	4.187	4.026	4.041	4.265	41.074
	3.2 M	3.952	3.935	3.948	3.979	3.879	4.041	3.889	3.938	4.112	3.891	39.564
	6.4 M	3.871	3.950	3.912	3.931	3.855	4.043	3.846	4.048	3.868	3.870	39.194
	12.8 M	3.865	3.890	3.882	3.910	3.938	3.915	4.023	3.835	3.850	3.836	38.944

1 Column with dictionary of size 25K - Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	7.037	6.922	6.934	6.971	7.029	6.906	7.076	6.924	6.954	6.993	69.746
	200 K	5.699	5.716	5.692	5.674	5.755	5.685	5.804	5.633	5.644	5.853	57.155
	400 K	5.096	5.095	5.130	5.119	5.038	5.187	5.015	5.030	5.258	5.028	50.996
	800 K	4.794	4.741	4.799	4.816	4.848	4.745	4.708	4.915	4.688	4.810	47.864
	1.6 M	4.666	4.673	4.669	4.613	4.670	4.711	4.743	4.597	4.594	4.818	46.754
	3.2 M	4.534	4.506	4.535	4.551	4.445	4.622	4.450	4.525	4.705	4.468	45.341
	6.4 M	4.530	4.509	4.477	4.496	4.422	4.617	4.428	4.601	4.426	4.456	44.962
	12.8 M	4.502	4.457	4.434	4.460	4.500	4.492	4.590	4.385	4.385	4.386	44.591

50,000 values in dictionary

No Buffer

1 Column with dictionary of size 50K - Insertions											
		Size Delta 2									Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	0 K	5.803	5.879	5.763	5.728	5.828	5.712	5.689	5.853	5.693	5.712	57.660

1 Column with dictionary of size 50K - Merge											
		Size Delta 2									Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	0 K	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

1 Column with dictionary of size 50K - Total											
		Size Delta 2									Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	0 K	5.803	5.879	5.763	5.728	5.828	5.712	5.689	5.853	5.693	5.712	57.660

Index Buffer

1 Column with dictionary of size 50K - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	6.118	6.172	6.114	6.065	6.352	6.140	6.165	6.104	6.030	6.115	61.375
	200 K	6.082	6.114	6.165	6.076	6.050	6.076	6.081	6.311	6.095	6.107	61.157
	400 K	6.162	6.122	6.060	6.078	6.043	6.040	6.073	6.204	6.085	6.085	60.952
	800 K	6.103	6.044	6.081	6.073	6.100	6.097	6.269	6.077	6.261	6.423	61.528
	1.6 M	6.120	6.070	6.088	6.112	6.083	6.195	6.111	6.245	6.295	6.256	61.575
	3.2 M	6.076	6.080	6.070	6.069	6.153	6.097	6.093	6.052	6.101	6.074	60.865
	6.4 M	6.226	6.065	6.112	6.095	6.170	6.163	6.123	6.121	6.086	6.088	61.249
	12.8 M	6.145	6.043	6.236	6.290	6.032	6.058	6.110	6.110	6.092	6.056	61.172

1 Column with dictionary of size 50K - Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	0.091	0.068	0.079	0.102	0.143	0.033	0.183	0.029	0.032	0.032	0.792
	200 K	0.088	0.091	0.095	0.027	0.119	0.035	0.166	0.028	0.026	0.258	0.933
	400 K	0.078	0.099	0.088	0.114	0.031	0.159	0.029	0.033	0.212	0.028	0.871
	800 K	0.090	0.060	0.077	0.103	0.149	0.030	0.032	0.205	0.030	0.028	0.804
	1.6 M	0.076	0.105	0.097	0.030	0.137	0.030	0.177	0.030	0.031	0.278	0.991
	3.2 M	0.066	0.090	0.091	0.127	0.029	0.173	0.030	0.029	0.249	0.030	0.914
	6.4 M	0.043	0.092	0.078	0.099	0.032	0.162	0.030	0.227	0.030	0.031	0.824
	12.8 M	0.030	0.055	0.082	0.105	0.131	0.030	0.168	0.030	0.030	0.030	0.691

1 Column with dictionary of size 50K - Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	6.209	6.240	6.193	6.167	6.495	6.173	6.348	6.133	6.062	6.147	62.167
	200 K	6.170	6.205	6.260	6.103	6.169	6.111	6.247	6.339	6.121	6.365	62.090
	400 K	6.240	6.221	6.148	6.192	6.074	6.199	6.102	6.237	6.297	6.113	61.823
	800 K	6.193	6.104	6.158	6.176	6.249	6.127	6.301	6.282	6.291	6.451	62.332
	1.6 M	6.196	6.175	6.185	6.142	6.220	6.225	6.288	6.275	6.326	6.534	62.566
	3.2 M	6.142	6.170	6.161	6.196	6.182	6.270	6.123	6.081	6.350	6.104	61.779
	6.4 M	6.269	6.157	6.190	6.194	6.202	6.325	6.153	6.348	6.116	6.119	62.073
	12.8 M	6.175	6.098	6.318	6.395	6.163	6.088	6.278	6.140	6.122	6.086	61.863

Standard Buffer

1 Column with dictionary of size 50K - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	0.560	0.582	0.526	0.565	0.535	0.533	0.563	0.564	0.563	0.533	5.524
	200 K	0.562	0.573	0.557	0.540	0.570	0.541	0.560	0.576	0.556	0.551	5.586
	400 K	0.550	0.567	0.569	0.548	0.550	0.573	0.537	0.558	0.559	0.535	5.546
	800 K	0.586	0.576	0.541	0.566	0.557	0.574	0.589	0.593	0.574	0.550	5.706
	1.6 M	0.643	0.543	0.565	0.565	0.575	0.572	0.567	0.557	0.583	0.576	5.746
	3.2 M	0.582	0.575	0.540	0.574	0.572	0.565	0.572	0.596	0.580	0.592	5.748
	6.4 M	0.574	0.556	0.594	0.586	0.534	0.585	0.557	0.587	0.577	0.573	5.723
	12.8 M	0.661	0.569	0.568	0.581	0.575	0.547	0.575	0.571	0.587	0.560	5.794

1 Column with dictionary of size 50K - Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	10.208	9.955	9.970	10.000	10.102	10.109	10.010	10.194	10.568	9.906	101.022
	200 K	7.453	7.207	7.302	7.260	7.295	7.184	7.217	7.371	7.190	7.220	72.699
	400 K	5.810	5.701	5.797	5.767	5.816	5.675	5.689	5.965	5.720	5.674	57.614
	800 K	5.040	5.024	4.971	5.031	5.034	4.925	4.944	5.105	4.923	5.002	49.999
	1.6 M	4.821	4.640	4.602	4.625	4.676	4.575	4.548	4.768	4.543	4.631	46.429
	3.2 M	4.491	4.402	4.413	4.477	4.489	4.376	4.375	4.593	4.381	4.393	44.390
	6.4 M	4.415	4.353	4.419	4.365	4.399	4.300	4.276	4.505	4.314	4.289	43.635
	12.8 M	4.407	4.304	4.279	4.346	4.350	4.246	4.253	4.528	4.248	4.285	43.246

1 Column with dictionary of size 50K - Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	10.768	10.537	10.496	10.565	10.637	10.642	10.573	10.758	11.131	10.439	106.546
	200 K	8.015	7.780	7.859	7.800	7.865	7.725	7.777	7.947	7.746	7.771	78.285
	400 K	6.360	6.268	6.366	6.315	6.366	6.248	6.226	6.523	6.279	6.209	63.160
	800 K	5.626	5.600	5.512	5.597	5.591	5.499	5.533	5.698	5.497	5.552	55.705
	1.6 M	5.464	5.183	5.167	5.190	5.251	5.147	5.115	5.325	5.126	5.207	52.175
	3.2 M	5.073	4.977	4.953	5.051	5.061	4.941	4.947	5.189	4.961	4.985	50.138
	6.4 M	4.989	4.909	5.013	4.951	4.933	4.885	4.833	5.092	4.891	4.862	49.358
	12.8 M	5.068	4.873	4.847	4.927	4.925	4.793	4.828	5.099	4.835	4.845	49.040

Sort Buffer

1 Column with dictionary of size 50K - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	0.803	0.802	0.827	0.798	0.800	0.780	0.824	0.807	0.842	0.828	8.111
	200 K	0.823	0.814	0.859	0.800	0.823	0.798	0.803	0.794	0.797	0.815	8.126
	400 K	0.839	0.809	0.820	0.783	0.820	0.811	0.829	0.805	0.809	0.807	8.132
	800 K	0.828	0.873	0.812	0.826	0.806	0.837	0.813	0.815	0.797	0.812	8.219
	1.6 M	0.789	0.809	0.838	0.812	0.807	0.786	0.816	0.811	0.807	0.813	8.088
	3.2 M	0.860	0.802	0.819	0.791	0.808	0.817	0.813	0.800	0.791	0.822	8.123
	6.4 M	0.892	0.797	0.819	0.804	0.822	0.826	0.817	0.842	0.815	0.805	8.239
	12.8 M	0.970	0.803	0.813	0.795	0.776	0.776	0.779	0.776	0.860	0.809	8.157

1 Column with dictionary of size 50K - Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	6.148	5.931	5.936	5.949	6.018	5.877	5.878	6.058	5.894	5.885	59.574
	200 K	5.794	5.565	6.129	5.671	5.727	5.528	5.527	5.716	5.537	5.532	56.726
	400 K	5.925	5.703	5.723	5.747	5.926	5.731	5.660	5.875	6.040	5.707	58.037
	800 K	5.948	6.311	5.644	5.667	5.761	5.964	5.595	5.813	5.619	5.582	57.904
	1.6 M	6.476	6.144	6.185	6.453	6.243	6.030	6.019	6.205	6.056	6.028	61.839
	3.2 M	7.160	6.117	6.324	6.169	6.193	6.067	6.069	6.291	6.066	6.573	63.029
	6.4 M	8.177	6.611	8.455	6.649	6.684	6.579	6.599	6.762	6.577	6.629	69.722
	12.8 M	9.906	6.758	6.738	6.789	6.827	6.685	6.690	6.897	6.703	6.687	70.680

1 Column with dictionary of size 50K - Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	6.951	6.733	6.763	6.747	6.818	6.657	6.702	6.865	6.736	6.713	67.685
	200 K	6.617	6.379	6.988	6.471	6.550	6.326	6.330	6.510	6.334	6.347	64.852
	400 K	6.764	6.512	6.543	6.530	6.746	6.542	6.489	6.680	6.849	6.514	66.169
	800 K	6.776	7.184	6.456	6.493	6.567	6.801	6.408	6.628	6.416	6.394	66.123
	1.6 M	7.265	6.953	7.023	7.265	7.050	6.816	6.835	7.016	6.863	6.841	69.927
	3.2 M	8.020	6.919	7.143	6.960	7.001	6.884	6.882	7.091	6.857	7.395	71.152
	6.4 M	9.069	7.408	9.274	7.453	7.506	7.405	7.416	7.604	7.392	7.434	77.961
	12.8 M	10.876	7.561	7.551	7.584	7.603	7.461	7.469	7.673	7.563	7.496	78.837

Map Buffer

1 Column with dictionary of size 50K - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	0.545	0.565	0.560	0.548	0.588	0.563	0.575	0.571	0.566	0.578	5.659
	200 K	0.564	0.554	0.579	0.557	0.562	0.561	0.550	0.585	0.560	0.574	5.646
	400 K	0.574	0.563	0.543	0.553	0.556	0.546	0.572	0.597	0.522	0.572	5.598
	800 K	0.581	0.546	0.565	0.570	0.552	0.590	0.566	0.566	0.570	0.558	5.664
	1.6 M	0.556	0.594	0.584	0.589	0.572	0.562	0.582	0.581	0.591	0.557	5.768
	3.2 M	0.585	0.591	0.593	0.592	0.582	0.555	0.580	0.582	0.574	0.563	5.797
	6.4 M	0.592	0.597	0.581	0.550	0.570	0.572	0.574	0.575	0.560	0.566	5.737
	12.8 M	0.643	0.564	0.569	0.570	0.569	0.554	0.593	0.585	0.577	0.554	5.778

1 Column with dictionary of size 50K - Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	10.146	10.108	10.101	10.204	10.142	10.073	10.200	10.012	10.162	10.146	101.294
	200 K	7.508	7.274	7.321	7.212	7.302	7.261	7.453	7.189	7.244	7.423	73.187
	400 K	5.896	5.888	5.909	5.906	5.839	6.037	5.820	5.818	6.032	5.823	58.968
	800 K	5.194	5.136	5.160	5.216	5.234	5.109	5.100	5.331	5.115	5.148	51.743
	1.6 M	4.821	4.816	4.836	4.848	4.877	4.729	4.897	4.757	4.743	4.982	48.306
	3.2 M	4.591	4.657	4.611	4.636	4.530	4.662	4.520	4.558	4.779	4.562	46.106
	6.4 M	4.486	4.480	4.466	4.509	4.413	4.558	4.437	4.655	4.472	4.513	44.989
	12.8 M	4.384	4.413	4.414	4.462	4.468	4.458	4.544	4.348	4.383	4.388	44.262

1 Column with dictionary of size 50K - Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	10.691	10.673	10.661	10.752	10.730	10.636	10.775	10.583	10.728	10.724	106.953
	200 K	8.072	7.828	7.900	7.769	7.864	7.822	8.003	7.774	7.804	7.997	78.833
	400 K	6.470	6.451	6.452	6.459	6.395	6.583	6.392	6.415	6.554	6.395	64.566
	800 K	5.775	5.682	5.725	5.786	5.786	5.699	5.666	5.897	5.685	5.706	57.407
	1.6 M	5.377	5.410	5.420	5.437	5.449	5.291	5.479	5.338	5.334	5.539	54.074
	3.2 M	5.176	5.248	5.204	5.228	5.112	5.217	5.100	5.140	5.353	5.125	51.903
	6.4 M	5.078	5.077	5.047	5.059	4.983	5.130	5.011	5.230	5.032	5.079	50.726
	12.8 M	5.027	4.977	4.983	5.032	5.037	5.012	5.137	4.933	4.960	4.942	50.040

100,000 values in dictionary

No Buffer

1 Column with dictionary of size 100K - Insertions											
		Size Delta 2									Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	0 K	6.784	6.878	6.968	7.429	6.962	6.862	6.766	6.768	7.005	6.545	68.967

1 Column with dictionary of size 100K -Merge											
		Size Delta 2									Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	0 K	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

1 Column with dictionary of size 100K -Total											
		Size Delta 2									Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	0 K	6.784	6.878	6.968	7.429	6.962	6.862	6.766	6.768	7.005	6.545	68.967

Index Buffer

1 Column with dictionary of size 100K - Insertions											
		Size Delta 2									Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	7.055	7.079	7.062	6.914	6.924	6.913	6.896	6.820	6.927	6.931	69.521
	200 K	7.029	6.926	6.971	6.906	6.902	6.954	7.045	7.300	6.961	6.980	69.974
	400 K	7.042	6.932	6.881	6.835	7.199	6.947	6.949	6.858	6.880	6.879	69.402
	800 K	7.194	6.901	6.984	6.890	6.877	6.851	6.874	6.873	6.863	6.896	69.203
	1.6 M	7.102	7.097	6.864	6.960	6.990	6.999	6.907	6.887	6.867	6.847	69.520
	3.2 M	6.966	6.858	6.876	7.044	6.864	6.895	6.901	6.886	7.012	7.076	69.378
	6.4 M	6.961	6.957	6.837	6.954	6.933	6.926	6.944	7.025	7.246	7.033	69.816
	12.8 M	7.039	6.858	6.901	7.091	6.891	6.839	6.849	6.872	6.896	6.818	69.054

1 Column with dictionary of size 100K -Merge											
		Size Delta 2									Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	0.089	0.065	0.075	0.112	0.135	0.028	0.187	0.022	0.032	0.028	0.773
	200 K	0.078	0.091	0.085	0.028	0.123	0.025	0.171	0.030	0.024	0.252	0.907
	400 K	0.076	0.087	0.085	0.114	0.029	0.160	0.028	0.031	0.218	0.026	0.854
	800 K	0.097	0.062	0.076	0.104	0.149	0.029	0.029	0.195	0.029	0.028	0.798
	1.6 M	0.084	0.108	0.096	0.027	0.169	0.031	0.178	0.029	0.028	0.269	1.019
	3.2 M	0.066	0.089	0.091	0.125	0.030	0.175	0.029	0.031	0.254	0.031	0.921
	6.4 M	0.043	0.092	0.080	0.098	0.030	0.161	0.030	0.299	0.034	0.030	0.897
	12.8 M	0.030	0.056	0.080	0.105	0.130	0.029	0.173	0.029	0.029	0.030	0.691

1 Column with dictionary of size 100K -Total											
		Size Delta 2									Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	7.144	7.144	7.137	7.026	7.059	6.941	7.083	6.842	6.959	6.959	70.294
	200 K	7.107	7.017	7.056	6.934	7.025	6.979	7.216	7.330	6.985	7.232	70.881
	400 K	7.118	7.019	6.966	6.949	7.228	7.107	6.977	6.889	7.098	6.905	70.256
	800 K	7.291	6.963	7.060	6.994	7.026	6.880	6.903	7.068	6.892	6.924	70.001
	1.6 M	7.186	7.205	6.960	6.987	7.159	7.030	7.085	6.916	6.895	7.116	70.539
	3.2 M	7.032	6.947	6.967	7.169	6.894	7.070	6.930	6.917	7.266	7.107	70.299
	6.4 M	7.004	7.049	6.917	7.052	6.963	7.087	6.974	7.324	7.280	7.063	70.713
	12.8 M	7.069	6.914	6.981	7.196	7.021	6.868	7.022	6.901	6.925	6.848	69.745

Standard Buffer

1 Column with dictionary of size 100K - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	0.538	0.563	0.550	0.573	0.538	0.559	0.564	0.573	0.586	0.582	5.626
	200 K	0.567	0.545	0.566	0.529	0.565	0.590	0.557	0.563	0.539	0.573	5.594
	400 K	0.577	0.539	0.562	0.555	0.569	0.568	0.568	0.561	0.574	0.565	5.638
	800 K	0.590	0.602	0.555	0.587	0.567	0.577	0.552	0.589	0.546	0.570	5.735
	1.6 M	0.598	0.575	0.553	0.570	0.586	0.564	0.573	0.573	0.580	0.554	5.726
	3.2 M	0.587	0.568	0.567	0.583	0.602	0.569	0.592	0.570	0.571	0.573	5.782
	6.4 M	0.605	0.571	0.552	0.586	0.565	0.566	0.564	0.575	0.554	0.593	5.731
	12.8 M	0.670	0.568	0.527	0.568	0.569	0.569	0.577	0.553	0.558	0.558	5.717

1 Column with dictionary of size 100K -Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	18.792	18.817	18.826	18.793	18.879	18.863	19.173	18.883	19.115	18.661	188.802
	200 K	12.197	12.287	12.311	12.426	12.376	12.661	12.339	12.353	12.195	12.671	123.816
	400 K	8.817	8.786	8.751	8.865	8.797	8.699	8.728	8.891	8.705	8.733	87.772
	800 K	7.228	7.132	7.042	7.067	7.108	6.990	6.987	7.169	6.987	7.022	70.732
	1.6 M	6.401	6.199	6.209	6.245	6.271	6.152	6.254	6.410	6.214	6.213	62.568
	3.2 M	5.963	5.805	5.807	5.820	5.855	5.864	5.817	6.158	5.952	5.749	58.790
	6.4 M	5.705	5.780	5.625	5.630	5.650	5.559	5.562	5.760	5.572	5.559	56.402
	12.8 M	5.622	5.501	5.539	5.553	5.701	5.483	5.499	5.679	5.482	5.468	55.527

1 Column with dictionary of size 100K -Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	19.330	19.380	19.376	19.366	19.417	19.422	19.737	19.456	19.701	19.243	194.428
	200 K	12.764	12.832	12.877	12.955	12.941	13.251	12.896	12.916	12.734	13.244	129.410
	400 K	9.394	9.325	9.313	9.420	9.366	9.267	9.296	9.452	9.279	9.298	93.410
	800 K	7.818	7.734	7.597	7.654	7.675	7.567	7.539	7.758	7.533	7.592	76.467
	1.6 M	6.999	6.774	6.762	6.815	6.857	6.716	6.827	6.983	6.794	6.767	68.294
	3.2 M	6.550	6.373	6.374	6.403	6.457	6.433	6.409	6.728	6.523	6.322	64.572
	6.4 M	6.310	6.351	6.177	6.216	6.215	6.125	6.126	6.335	6.126	6.152	62.133
	12.8 M	6.292	6.069	6.066	6.121	6.270	6.052	6.076	6.232	6.040	6.026	61.244

Sort Buffer

1 Column with dictionary of size 100K - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	0.803	0.849	0.787	0.775	0.814	0.843	0.825	0.801	0.801	0.793	8.091
	200 K	0.838	0.817	0.818	0.807	0.792	0.838	0.821	0.812	0.834	0.775	8.152
	400 K	0.822	0.813	0.783	0.818	0.791	0.848	0.838	0.803	0.806	0.830	8.152
	800 K	0.815	0.810	0.826	0.797	0.776	0.800	0.812	0.797	0.813	0.833	8.079
	1.6 M	0.804	0.801	0.811	0.831	0.817	0.786	0.818	0.822	0.801	0.829	8.120
	3.2 M	0.854	0.803	0.831	0.798	0.799	0.818	0.803	0.799	0.800	0.812	8.117
	6.4 M	0.854	0.813	0.823	0.841	0.827	0.797	0.788	0.815	0.803	0.803	8.164
	12.8 M	0.962	0.804	0.808	0.788	0.781	0.797	0.808	0.788	0.813	0.781	8.130

1 Column with dictionary of size 100K -Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	8.170	7.951	7.971	8.014	8.001	7.885	7.914	8.083	7.967	7.899	79.855
	200 K	6.799	6.625	6.548	6.616	6.605	6.606	6.499	6.692	6.504	6.511	66.005
	400 K	6.400	6.163	6.458	6.569	6.268	6.141	6.139	6.392	6.607	6.863	64.000
	800 K	6.106	5.920	5.905	6.107	5.897	5.850	6.194	6.182	5.992	5.866	60.019
	1.6 M	6.538	6.101	6.114	6.140	6.182	6.062	6.069	6.229	6.093	6.604	62.132
	3.2 M	6.937	6.100	6.254	6.145	6.176	6.060	6.087	6.253	6.528	6.076	62.616
	6.4 M	8.252	6.587	6.630	6.617	6.668	6.535	6.583	6.707	6.558	6.570	67.707
	12.8 M	9.937	6.676	6.715	6.814	6.779	6.727	6.650	6.861	6.695	6.780	70.634

1 Column with dictionary of size 100K -Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	8.973	8.800	8.758	8.789	8.815	8.728	8.739	8.884	8.768	8.692	87.946
	200 K	7.637	7.442	7.366	7.423	7.397	7.444	7.320	7.504	7.338	7.286	74.157
	400 K	7.222	6.976	7.241	7.387	7.059	6.989	6.977	7.195	7.413	7.693	72.152
	800 K	6.921	6.730	6.731	6.904	6.673	6.650	7.006	6.979	6.805	6.699	68.098
	1.6 M	7.342	6.902	6.925	6.971	6.999	6.848	6.887	7.051	6.894	7.433	70.252
	3.2 M	7.791	6.903	7.085	6.943	6.975	6.878	6.890	7.052	7.328	6.888	70.733
	6.4 M	9.106	7.400	7.453	7.458	7.495	7.332	7.371	7.522	7.361	7.373	75.871
	12.8 M	10.899	7.480	7.523	7.602	7.560	7.524	7.458	7.649	7.508	7.561	78.764

Map Buffer

1 Column with dictionary of size 100K - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	0.545	0.582	0.563	0.587	0.560	0.552	0.537	0.568	0.556	0.556	5.606
	200 K	0.550	0.576	0.538	0.559	0.540	0.588	0.577	0.590	0.547	0.567	5.632
	400 K	0.586	0.570	0.571	0.576	0.569	0.580	0.567	0.574	0.556	0.557	5.706
	800 K	0.580	0.556	0.550	0.588	0.586	0.558	0.587	0.535	0.571	0.559	5.670
	1.6 M	0.570	0.597	0.559	0.603	0.566	0.601	0.581	0.570	0.530	0.553	5.730
	3.2 M	0.615	0.569	0.556	0.573	0.566	0.580	0.597	0.584	0.551	0.575	5.766
	6.4 M	0.599	0.591	0.580	0.556	0.569	0.580	0.552	0.567	0.532	0.552	5.678
	12.8 M	0.651	0.577	0.564	0.593	0.554	0.541	0.561	0.577	0.581	0.534	5.733

1 Column with dictionary of size 100K -Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	19.087	19.264	18.962	18.979	19.026	19.834	19.376	18.909	18.949	19.482	191.868
	200 K	12.394	12.907	12.857	12.298	12.371	12.281	12.425	12.467	12.606	12.511	125.117
	400 K	9.042	8.996	9.010	9.110	9.439	9.130	8.985	8.996	9.147	8.960	90.815
	800 K	7.358	7.275	7.306	7.385	7.534	7.261	7.262	7.511	7.274	7.273	73.439
	1.6 M	6.488	6.616	6.525	6.403	6.505	6.395	6.560	6.392	6.451	6.664	64.999
	3.2 M	6.058	5.995	5.996	6.043	5.919	6.059	5.927	6.448	6.247	5.943	60.635
	6.4 M	5.760	5.974	5.754	5.781	5.692	5.814	5.798	5.927	5.698	5.712	57.910
	12.8 M	5.646	5.613	5.792	5.658	5.715	5.608	5.761	5.753	5.578	5.795	56.919

1 Column with dictionary of size 100K -Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	19.632	19.846	19.525	19.566	19.586	20.386	19.913	19.477	19.505	20.038	197.474
	200 K	12.944	13.483	13.395	12.857	12.911	12.869	13.002	13.057	13.153	13.078	130.749
	400 K	9.628	9.566	9.581	9.686	10.008	9.710	9.552	9.570	9.703	9.517	96.521
	800 K	7.938	7.831	7.856	7.973	8.120	7.819	7.849	8.046	7.845	7.832	79.109
	1.6 M	7.058	7.213	7.084	7.006	7.071	6.996	7.141	6.962	6.981	7.217	70.729
	3.2 M	6.673	6.564	6.552	6.616	6.485	6.639	6.524	7.032	6.798	6.518	66.401
	6.4 M	6.359	6.565	6.334	6.337	6.261	6.394	6.350	6.494	6.230	6.264	63.588
	12.8 M	6.297	6.190	6.356	6.251	6.269	6.149	6.322	6.330	6.159	6.329	62.652

200,000 values in dictionary

No Buffer

1 Column with dictionary of size 200K - Insertions											
		Size Delta 2									Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	0 K	8.424	8.398	8.643	8.724	8.776	8.304	8.570	9.199	8.554	8.609	86.201

1 Column with dictionary of size 200K - Merge											
		Size Delta 2									Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	0 K	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

1 Column with dictionary of size 200K - Total											
		Size Delta 2									Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	0 K	8.424	8.398	8.643	8.724	8.776	8.304	8.570	9.199	8.554	8.609	86.201

Index Buffer

1 Column with dictionary of size 200K - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	8.524	8.495	8.695	8.349	8.575	8.348	8.528	8.333	8.623	8.563	85.033
	200 K	8.489	8.626	8.607	8.456	8.591	8.616	8.382	8.370	8.369	8.396	84.902
	400 K	8.734	8.364	8.332	8.455	8.331	8.333	8.341	8.343	8.363	8.797	84.393
	800 K	8.762	8.319	8.370	8.307	8.693	8.527	8.348	8.457	8.446	8.296	84.525
	1.6 M	8.686	8.316	8.305	8.306	8.277	8.325	8.333	8.458	8.559	8.317	83.882
	3.2 M	8.494	8.368	8.424	8.323	8.385	8.407	8.333	8.323	8.350	8.356	83.763
	6.4 M	8.576	8.707	8.667	8.492	8.787	8.740	8.417	8.315	8.382	8.437	85.520
	12.8 M	8.736	8.327	8.783	8.335	8.329	8.537	8.312	8.420	8.291	8.837	84.907

1 Column with dictionary of size 200K - Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	0.085	0.060	0.075	0.103	0.137	0.022	0.207	0.027	0.024	0.028	0.768
	200 K	0.080	0.100	0.090	0.027	0.144	0.028	0.173	0.032	0.029	0.231	0.934
	400 K	0.074	0.089	0.081	0.112	0.029	0.161	0.029	0.026	0.215	0.028	0.844
	800 K	0.096	0.058	0.080	0.104	0.151	0.032	0.031	0.196	0.030	0.027	0.805
	1.6 M	0.080	0.105	0.101	0.029	0.138	0.030	0.178	0.032	0.031	0.277	1.001
	3.2 M	0.072	0.089	0.111	0.124	0.029	0.171	0.031	0.028	0.249	0.029	0.933
	6.4 M	0.042	0.109	0.083	0.098	0.029	0.161	0.030	0.229	0.030	0.030	0.841
	12.8 M	0.030	0.055	0.081	0.107	0.132	0.029	0.168	0.031	0.030	0.029	0.692

1 Column with dictionary of size 200K - Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	8.609	8.555	8.770	8.452	8.712	8.370	8.735	8.360	8.647	8.591	85.801
	200 K	8.569	8.726	8.697	8.483	8.735	8.644	8.555	8.402	8.398	8.627	85.836
	400 K	8.808	8.453	8.413	8.567	8.360	8.494	8.370	8.369	8.578	8.825	85.237
	800 K	8.858	8.377	8.450	8.411	8.844	8.559	8.379	8.653	8.476	8.323	85.330
	1.6 M	8.766	8.421	8.406	8.335	8.415	8.355	8.511	8.490	8.590	8.594	84.883
	3.2 M	8.566	8.457	8.535	8.447	8.414	8.578	8.364	8.351	8.599	8.385	84.696
	6.4 M	8.618	8.816	8.750	8.590	8.816	8.901	8.447	8.544	8.412	8.467	86.361
	12.8 M	8.766	8.382	8.864	8.442	8.461	8.566	8.480	8.451	8.321	8.866	85.599

Standard Buffer

1 Column with dictionary of size 200K - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	0.562	0.556	0.553	0.559	0.554	0.544	0.571	0.580	0.570	0.582	5.631
	200 K	0.559	0.548	0.559	0.555	0.571	0.574	0.548	0.567	0.567	0.586	5.634
	400 K	0.547	0.547	0.546	0.553	0.545	0.554	0.564	0.541	0.564	0.545	5.506
	800 K	0.594	0.547	0.544	0.573	0.576	0.569	0.563	0.578	0.573	0.580	5.697
	1.6 M	0.571	0.587	0.579	0.574	0.550	0.561	0.580	0.588	0.597	0.547	5.734
	3.2 M	0.600	0.552	0.545	0.588	0.555	0.579	0.576	0.583	0.586	0.557	5.721
	6.4 M	0.637	0.590	0.569	0.563	0.567	0.551	0.586	0.565	0.569	0.557	5.754
	12.8 M	0.674	0.556	0.586	0.595	0.575	0.558	0.546	0.588	0.556	0.566	5.800

1 Column with dictionary of size 200K - Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	21.003	22.186	20.863	20.898	20.938	20.805	21.060	22.226	20.941	20.807	211.727
	200 K	23.891	23.818	23.745	23.418	23.470	23.349	23.350	23.832	23.353	23.414	235.640
	400 K	15.842	15.022	15.416	15.071	15.108	15.187	14.969	15.157	14.964	15.105	151.841
	800 K	11.475	11.213	11.359	11.012	10.983	10.862	10.878	11.120	11.516	10.986	111.404
	1.6 M	9.031	9.377	8.895	8.899	8.942	8.841	8.834	9.023	8.852	8.847	89.541
	3.2 M	8.555	8.095	8.408	8.003	8.829	8.815	7.929	8.111	7.951	7.926	82.622
	6.4 M	7.572	7.389	7.430	7.480	8.275	7.672	7.369	7.555	7.397	7.367	75.506
	12.8 M	8.137	7.153	7.150	7.184	7.278	7.122	7.122	7.617	7.144	7.148	73.055

1 Column with dictionary of size 200K - Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	21.565	22.742	21.416	21.457	21.492	21.349	21.631	22.806	21.511	21.389	217.358
	200 K	24.450	24.366	24.304	23.973	24.041	23.923	23.898	24.399	23.920	24.000	241.274
	400 K	16.389	15.569	15.962	15.624	15.653	15.741	15.533	15.698	15.528	15.650	157.347
	800 K	12.069	11.760	11.903	11.585	11.559	11.431	11.441	11.698	12.089	11.566	117.101
	1.6 M	9.602	9.964	9.474	9.473	9.492	9.402	9.414	9.611	9.449	9.394	95.275
	3.2 M	9.155	8.647	8.953	8.591	9.384	9.394	8.505	8.694	8.537	8.483	88.343
	6.4 M	8.209	7.979	7.999	8.043	8.842	8.223	7.955	8.120	7.966	7.924	81.260
	12.8 M	8.811	7.709	7.736	7.779	7.853	7.680	7.668	8.205	7.700	7.714	78.855

Sort Buffer

1 Column with dictionary of size 200K - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	0.836	0.790	0.801	0.794	0.833	0.803	0.844	0.828	0.780	0.806	8.115
	200 K	0.798	0.803	0.800	0.786	0.806	0.829	0.806	0.820	0.834	0.832	8.114
	400 K	0.808	0.810	0.813	0.825	0.794	0.790	0.810	0.780	0.813	0.827	8.070
	800 K	0.812	0.814	0.805	0.807	0.818	0.814	0.816	0.840	0.823	0.809	8.158
	1.6 M	0.821	0.814	0.822	0.796	0.820	0.803	0.806	0.826	0.833	0.822	8.163
	3.2 M	0.826	0.809	0.844	0.803	0.806	0.814	0.800	0.820	0.827	0.798	8.147
	6.4 M	0.851	0.810	0.798	0.793	0.799	0.796	0.830	0.797	0.814	0.817	8.105
	12.8 M	0.936	0.827	0.812	0.797	0.796	0.821	0.788	0.812	0.796	0.785	8.170

1 Column with dictionary of size 200K - Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	8.507	8.190	8.200	8.229	8.307	8.185	8.160	8.329	8.187	8.168	82.462
	200 K	8.721	8.532	8.508	8.513	8.700	8.441	8.456	8.650	8.458	8.457	85.436
	400 K	7.354	7.552	7.284	7.161	7.234	7.241	7.299	7.248	7.124	7.059	72.556
	800 K	6.663	6.267	6.499	6.488	6.312	6.213	6.199	6.387	6.240	6.447	63.715
	1.6 M	6.730	6.272	6.543	6.329	6.491	6.241	6.285	6.993	6.279	6.272	64.435
	3.2 M	7.022	6.146	6.163	6.178	6.260	6.115	6.104	6.318	6.102	6.104	62.512
	6.4 M	9.019	7.144	6.598	6.636	6.652	6.896	6.545	6.740	6.584	6.532	69.346
	12.8 M	10.014	6.662	6.673	6.694	6.728	7.221	6.620	6.813	7.453	6.613	71.491

1 Column with dictionary of size 200K - Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	9.343	8.980	9.001	9.023	9.140	8.988	9.004	9.157	8.967	8.974	90.577
	200 K	9.519	9.335	9.308	9.299	9.506	9.270	9.262	9.470	9.292	9.289	93.550
	400 K	8.162	8.362	8.097	7.986	8.028	8.031	8.109	8.028	7.937	7.886	80.626
	800 K	7.475	7.081	7.304	7.295	7.130	7.027	7.015	7.227	7.063	7.256	71.873
	1.6 M	7.551	7.086	7.365	7.125	7.311	7.044	7.091	7.819	7.112	7.094	72.598
	3.2 M	7.848	6.955	7.007	6.981	7.066	6.929	6.904	7.138	6.929	6.902	70.659
	6.4 M	9.870	7.954	7.396	7.429	7.451	7.692	7.375	7.537	7.398	7.349	77.451
	12.8 M	10.950	7.489	7.485	7.491	7.524	8.042	7.408	7.625	8.249	7.398	79.661

Map Buffer

1 Column with dictionary of size 200K - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	0.582	0.563	0.582	0.581	0.576	0.563	0.545	0.578	0.574	0.594	5.738
	200 K	0.536	0.583	0.570	0.552	0.503	0.553	0.542	0.547	0.577	0.559	5.522
	400 K	0.535	0.543	0.548	0.569	0.568	0.573	0.552	0.532	0.558	0.579	5.557
	800 K	0.597	0.532	0.572	0.547	0.556	0.562	0.548	0.590	0.573	0.585	5.662
	1.6 M	0.594	0.571	0.564	0.581	0.594	0.582	0.583	0.580	0.555	0.567	5.771
	3.2 M	0.595	0.566	0.556	0.585	0.575	0.592	0.579	0.571	0.553	0.578	5.750
	6.4 M	0.641	0.584	0.547	0.574	0.583	0.601	0.543	0.583	0.583	0.590	5.829
	12.8 M	0.681	0.585	0.600	0.581	0.567	0.565	0.576	0.573	0.564	0.548	5.840

1 Column with dictionary of size 200K - Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	21.294	21.779	21.489	22.127	21.212	21.068	21.729	21.073	21.096	21.109	213.976
	200 K	23.783	25.547	23.967	24.313	23.758	24.886	24.185	23.706	23.693	23.934	241.772
	400 K	15.535	16.061	15.627	16.469	15.355	15.611	15.353	16.324	15.602	15.390	157.327
	800 K	11.562	11.543	11.621	11.286	11.697	11.203	11.357	11.412	11.226	11.327	114.234
	1.6 M	9.551	9.199	9.308	9.174	9.583	9.150	9.560	9.199	9.148	9.395	93.267
	3.2 M	8.207	8.381	8.119	8.169	8.044	8.589	8.042	8.435	8.747	8.039	82.772
	6.4 M	7.854	7.660	7.623	7.609	7.671	7.665	7.698	7.772	7.539	7.557	76.648
	12.8 M	7.356	7.266	7.315	7.354	7.754	7.394	7.422	7.261	7.293	7.364	73.779

1 Column with dictionary of size 200K - Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	21.876	22.342	22.071	22.708	21.788	21.631	22.274	21.651	21.670	21.703	219.714
	200 K	24.319	26.130	24.537	24.865	24.261	25.439	24.727	24.253	24.270	24.493	247.294
	400 K	16.070	16.604	16.175	17.038	15.923	16.184	15.905	16.856	16.160	15.969	162.884
	800 K	12.159	12.075	12.193	11.833	12.253	11.765	11.905	12.002	11.799	11.912	119.896
	1.6 M	10.145	9.770	9.872	9.755	10.177	9.732	10.143	9.779	9.703	9.962	99.038
	3.2 M	8.802	8.947	8.675	8.754	8.619	9.181	8.621	9.006	9.300	8.617	88.522
	6.4 M	8.495	8.244	8.170	8.183	8.254	8.266	8.241	8.355	8.122	8.147	82.477
	12.8 M	8.037	7.851	7.915	7.935	8.321	7.959	7.998	7.834	7.857	7.912	79.619

400,000 values in dictionary

No Buffer

1 Column with dictionary of size 400K - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	0 K	11.640	11.687	11.364	10.776	10.935	10.714	10.760	11.013	10.855	10.866	110.610

1 Column with dictionary of size 400K - Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	0 K	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

1 Column with dictionary of size 400K - Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	0 K	11.640	11.687	11.364	10.776	10.935	10.714	10.760	11.013	10.855	10.866	110.610

Index Buffer

1 Column with dictionary of size 400K - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	11.461	11.643	10.833	10.877	11.205	10.990	11.623	10.814	10.821	10.834	111.101
	200 K	11.955	10.842	10.809	10.832	10.846	11.342	10.817	10.845	10.800	10.954	110.042
	400 K	11.759	10.855	10.830	10.870	11.732	10.855	10.866	10.840	10.911	10.829	110.347
	800 K	11.147	10.806	11.092	12.335	11.448	11.116	11.146	10.748	10.800	11.841	112.479
	1.6 M	11.406	10.768	10.788	10.799	10.802	10.838	10.988	10.813	10.808	10.796	108.806
	3.2 M	11.457	10.768	11.101	11.155	11.671	10.756	10.753	10.791	10.810	10.782	110.044
	6.4 M	11.799	11.898	11.313	10.874	10.927	10.797	10.810	10.810	10.772	10.869	110.869
	12.8 M	11.094	10.800	10.781	10.755	10.838	11.434	10.854	10.790	10.797	10.788	108.931

1 Column with dictionary of size 400K - Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	0.095	0.062	0.083	0.092	0.165	0.039	0.199	0.036	0.032	0.024	0.827
	200 K	0.076	0.090	0.090	0.031	0.120	0.031	0.182	0.030	0.032	0.260	0.942
	400 K	0.081	0.090	0.084	0.114	0.031	0.160	0.029	0.032	0.220	0.029	0.870
	800 K	0.090	0.061	0.079	0.124	0.181	0.032	0.026	0.199	0.030	0.030	0.852
	1.6 M	0.080	0.102	0.100	0.030	0.138	0.027	0.189	0.031	0.029	0.249	0.975
	3.2 M	0.065	0.088	0.090	0.126	0.030	0.171	0.029	0.029	0.258	0.030	0.916
	6.4 M	0.050	0.107	0.084	0.100	0.029	0.164	0.030	0.235	0.030	0.030	0.859
	12.8 M	0.029	0.056	0.080	0.108	0.132	0.030	0.171	0.030	0.030	0.030	0.696

1 Column with dictionary of size 400K - Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	11.556	11.705	10.916	10.969	11.370	11.029	11.822	10.850	10.853	10.858	111.928
	200 K	12.031	10.932	10.899	10.863	10.966	11.373	10.999	10.875	10.832	11.214	110.984
	400 K	11.840	10.945	10.914	10.984	11.763	11.015	10.895	10.872	11.131	10.858	111.217
	800 K	11.237	10.867	11.171	12.459	11.629	11.148	11.172	10.947	10.830	11.871	113.331
	1.6 M	11.486	10.870	10.888	10.829	10.940	10.865	11.177	10.844	10.837	11.045	109.781
	3.2 M	11.522	10.856	11.191	11.281	11.701	10.927	10.782	10.820	11.068	10.812	110.960
	6.4 M	11.849	12.005	11.397	10.974	10.956	10.961	10.840	11.045	10.802	10.899	111.728
	12.8 M	11.123	10.856	10.861	10.863	10.970	11.464	11.025	10.820	10.827	10.818	109.627

Standard Buffer

1 Column with dictionary of size 400K - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	0.594	0.562	0.572	0.563	0.565	0.608	0.551	0.578	0.553	0.558	5.704
	200 K	0.548	0.551	0.573	0.551	0.588	0.583	0.580	0.547	0.545	0.532	5.598
	400 K	0.557	0.551	0.542	0.564	0.582	0.567	0.564	0.544	0.561	0.598	5.630
	800 K	0.577	0.557	0.564	0.564	0.555	0.572	0.560	0.569	0.532	0.581	5.631
	1.6 M	0.602	0.565	0.570	0.573	0.559	0.559	0.583	0.590	0.588	0.545	5.734
	3.2 M	0.600	0.558	0.582	0.598	0.570	0.564	0.570	0.542	0.560	0.589	5.733
	6.4 M	0.627	0.577	0.558	0.556	0.575	0.606	0.592	0.564	0.601	0.565	5.821
	12.8 M	0.657	0.532	0.587	0.546	0.547	0.552	0.561	0.573	0.571	0.579	5.705

1 Column with dictionary of size 400K - Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	23.853	23.486	23.497	23.645	23.606	25.329	23.753	23.655	24.668	23.532	239.024
	200 K	26.482	27.393	25.805	25.816	25.868	26.814	27.461	27.174	27.518	25.917	266.248
	400 K	28.492	28.278	27.741	29.154	27.798	27.707	29.812	27.861	27.648	28.876	283.367
	800 K	18.438	18.383	18.123	18.237	18.196	18.077	18.590	18.278	18.475	18.653	183.450
	1.6 M	13.561	13.228	13.233	13.351	13.301	13.186	13.181	13.825	14.277	13.210	134.353
	3.2 M	11.273	10.984	10.966	10.985	11.038	10.930	10.917	11.187	10.940	10.953	110.173
	6.4 M	10.045	9.826	9.734	9.720	9.808	9.632	9.633	10.589	10.299	9.648	98.934
	12.8 M	9.513	9.135	9.143	9.225	9.361	9.252	9.102	9.333	9.096	9.092	92.252

1 Column with dictionary of size 400K - Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	24.447	24.048	24.069	24.208	24.171	25.937	24.304	24.233	25.221	24.090	244.728
	200 K	27.030	27.944	26.378	26.367	26.456	27.397	28.041	27.721	28.063	26.449	271.846
	400 K	29.049	28.829	28.283	29.718	28.380	28.274	30.376	28.405	28.209	29.474	288.997
	800 K	19.015	18.940	18.687	18.801	18.751	18.649	19.150	18.847	19.007	19.234	189.081
	1.6 M	14.163	13.793	13.803	13.924	13.860	13.745	13.764	14.415	14.865	13.755	140.087
	3.2 M	11.873	11.542	11.548	11.583	11.608	11.494	11.487	11.729	11.500	11.542	115.906
	6.4 M	10.672	10.403	10.292	10.276	10.383	10.238	10.225	11.153	10.900	10.213	104.755
	12.8 M	10.170	9.667	9.730	9.771	9.908	9.804	9.663	9.906	9.667	9.671	97.957

Sort Buffer

1 Column with dictionary of size 400K - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	0.791	0.818	0.811	0.806	0.819	0.806	0.859	0.812	0.791	0.800	8.113
	200 K	0.816	0.817	0.826	0.835	0.803	0.841	0.784	0.817	0.784	0.798	8.121
	400 K	0.819	0.809	0.816	0.778	0.808	0.822	0.806	0.827	0.800	0.869	8.154
	800 K	0.819	0.806	0.817	0.811	0.813	0.867	0.854	0.821	0.840	0.803	8.251
	1.6 M	0.839	0.811	0.795	0.818	0.808	0.798	0.808	0.807	0.812	0.820	8.116
	3.2 M	0.843	0.823	0.818	0.820	0.805	0.830	0.803	0.808	0.827	0.787	8.164
	6.4 M	0.855	0.811	0.811	0.797	0.824	0.809	0.812	0.808	0.812	0.815	8.154
	12.8 M	0.904	0.826	0.802	0.799	0.799	0.807	0.823	0.848	0.825	0.823	8.256

1 Column with dictionary of size 400K - Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	8.898	8.556	8.586	8.615	8.620	8.504	8.667	8.683	8.510	8.493	86.132
	200 K	9.031	8.978	8.807	8.762	8.801	8.800	8.731	8.859	8.686	8.684	88.139
	400 K	10.141	9.190	9.318	9.354	9.405	9.160	9.147	9.367	9.188	9.188	93.458
	800 K	7.556	7.280	7.297	7.324	7.471	7.260	7.245	7.421	7.265	7.256	73.375
	1.6 M	7.411	6.959	6.784	6.814	6.854	6.790	6.731	6.913	6.788	6.737	68.781
	3.2 M	7.261	6.331	6.366	6.388	6.438	6.301	6.309	6.771	6.309	6.317	64.791
	6.4 M	9.182	6.641	6.645	6.654	6.726	6.578	6.606	7.126	6.600	6.632	69.390
	12.8 M	10.610	6.633	6.680	6.684	6.722	6.598	7.622	8.179	6.725	6.822	73.275

1 Column with dictionary of size 400K - Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	9.689	9.374	9.397	9.421	9.439	9.310	9.526	9.495	9.301	9.293	94.245
	200 K	9.847	9.795	9.633	9.597	9.604	9.641	9.515	9.676	9.470	9.482	96.260
	400 K	10.960	9.999	10.134	10.132	10.213	9.982	9.953	10.194	9.988	10.057	101.612
	800 K	8.375	8.086	8.114	8.135	8.284	8.127	8.099	8.242	8.105	8.059	81.626
	1.6 M	8.250	7.770	7.579	7.632	7.662	7.588	7.539	7.720	7.600	7.557	76.897
	3.2 M	8.104	7.154	7.184	7.208	7.243	7.131	7.112	7.579	7.136	7.104	72.955
	6.4 M	10.037	7.452	7.456	7.451	7.550	7.387	7.418	7.934	7.412	7.447	77.544
	12.8 M	11.514	7.459	7.482	7.483	7.521	7.405	8.445	9.027	7.550	7.645	81.531

Map Buffer

1 Column with dictionary of size 400K - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	0.588	0.562	0.591	0.565	0.561	0.549	0.569	0.581	0.577	0.539	5.682
	200 K	0.560	0.538	0.554	0.566	0.573	0.528	0.551	0.565	0.533	0.590	5.558
	400 K	0.553	0.554	0.561	0.540	0.578	0.557	0.548	0.558	0.550	0.561	5.560
	800 K	0.588	0.581	0.562	0.585	0.570	0.572	0.564	0.584	0.562	0.570	5.738
	1.6 M	0.575	0.604	0.547	0.555	0.563	0.574	0.553	0.560	0.563	0.565	5.659
	3.2 M	0.614	0.572	0.584	0.565	0.569	0.569	0.595	0.602	0.590	0.564	5.824
	6.4 M	0.624	0.599	0.570	0.559	0.557	0.582	0.549	0.574	0.584	0.557	5.755
	12.8 M	0.681	0.569	0.572	0.560	0.585	0.589	0.562	0.572	0.558	0.570	5.818

1 Column with dictionary of size 400K - Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	23.971	23.661	23.663	23.681	23.728	24.310	23.822	23.638	23.860	23.657	237.991
	200 K	27.810	26.193	26.576	26.307	26.165	26.421	26.355	27.362	27.388	26.961	267.538
	400 K	28.690	29.560	28.390	28.411	28.500	28.480	28.356	30.027	28.855	28.407	287.676
	800 K	19.671	19.280	19.984	18.626	19.079	18.493	18.489	19.190	18.526	18.519	189.857
	1.6 M	14.156	14.296	13.669	13.591	13.706	13.667	13.770	13.608	13.933	14.213	138.609
	3.2 M	11.412	11.106	11.344	11.155	11.067	11.229	11.088	11.446	11.316	11.109	112.272
	6.4 M	10.180	9.920	9.905	9.936	9.864	10.002	9.849	10.082	9.894	9.856	99.488
	12.8 M	10.297	9.393	9.300	9.291	9.349	9.235	9.398	9.217	9.204	9.223	93.907

1 Column with dictionary of size 400K - Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	24.559	24.223	24.254	24.246	24.289	24.859	24.391	24.219	24.437	24.196	243.673
	200 K	28.370	26.731	27.130	26.873	26.738	26.949	26.906	27.927	27.921	27.551	273.096
	400 K	29.243	30.114	28.951	28.951	29.078	29.037	28.904	30.585	29.405	28.968	293.236
	800 K	20.259	19.861	20.546	19.211	19.649	19.065	19.053	19.774	19.088	19.089	195.595
	1.6 M	14.731	14.900	14.216	14.146	14.269	14.241	14.323	14.168	14.496	14.778	144.268
	3.2 M	12.026	11.678	11.928	11.720	11.636	11.798	11.683	12.048	11.906	11.673	118.096
	6.4 M	10.804	10.519	10.475	10.495	10.421	10.584	10.398	10.656	10.478	10.413	105.243
	12.8 M	10.978	9.962	9.872	9.851	9.934	9.824	9.960	9.789	9.762	9.793	99.725

800,000 values in dictionary

No Buffer

1 Column with dictionary of size 800K - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	0 K	14.213	12.815	14.045	13.502	13.794	13.632	12.557	12.642	12.446	12.474	132.120

1 Column with dictionary of size 800K - Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	0 K	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

1 Column with dictionary of size 800K - Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	0 K	14.213	12.815	14.045	13.502	13.794	13.632	12.557	12.642	12.446	12.474	132.120

Index Buffer

1 Column with dictionary of size 800K - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	13.619	12.944	12.982	12.955	12.915	12.890	12.883	13.928	13.407	14.839	133.362
	200 K	13.535	13.855	12.859	13.452	13.358	12.866	13.086	13.386	14.228	13.726	134.351
	400 K	14.171	14.069	12.861	13.842	13.710	12.871	13.548	12.836	12.823	12.795	133.526
	800 K	13.475	12.824	13.593	13.215	12.827	13.202	13.961	12.856	13.739	12.939	132.631
	1.6 M	15.326	13.119	13.100	12.830	13.025	12.868	12.872	13.553	16.750	14.114	137.557
	3.2 M	13.773	13.113	14.292	12.890	12.872	12.800	12.824	12.897	12.819	12.855	131.135
	6.4 M	13.587	12.961	13.359	13.386	13.208	13.638	13.043	13.129	15.538	12.834	134.683
	12.8 M	14.790	13.109	13.147	13.179	12.901	12.932	13.220	12.889	13.443	13.530	133.140

1 Column with dictionary of size 800K - Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	0.084	0.060	0.088	0.103	0.129	0.022	0.192	0.025	0.027	0.033	0.763
	200 K	0.081	0.111	0.087	0.031	0.146	0.027	0.175	0.037	0.029	0.378	1.102
	400 K	0.079	0.100	0.084	0.117	0.029	0.158	0.028	0.029	0.218	0.028	0.870
	800 K	0.088	0.062	0.097	0.105	0.153	0.028	0.031	0.193	0.031	0.032	0.820
	1.6 M	0.080	0.103	0.098	0.031	0.141	0.030	0.181	0.031	0.035	0.293	1.023
	3.2 M	0.065	0.091	0.123	0.127	0.031	0.179	0.030	0.030	0.233	0.031	0.940
	6.4 M	0.042	0.094	0.079	0.108	0.032	0.148	0.030	0.299	0.035	0.030	0.897
	12.8 M	0.030	0.074	0.097	0.107	0.132	0.030	0.193	0.030	0.030	0.029	0.752

1 Column with dictionary of size 800K - Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	13.703	13.004	13.070	13.058	13.044	12.912	13.075	13.953	13.434	14.872	134.125
	200 K	13.616	13.966	12.946	13.483	13.504	12.893	13.261	13.423	14.257	14.104	135.453
	400 K	14.250	14.169	12.945	13.959	13.739	13.029	13.576	12.865	13.041	12.823	134.396
	800 K	13.563	12.886	13.690	13.320	12.980	13.230	13.992	13.049	13.770	12.971	133.451
	1.6 M	15.406	13.222	13.198	12.861	13.166	12.898	13.053	13.584	16.785	14.407	138.580
	3.2 M	13.838	13.204	14.415	13.017	12.903	12.979	12.854	12.927	13.052	12.886	132.075
	6.4 M	13.629	13.055	13.438	13.494	13.240	13.786	13.073	13.428	15.573	12.864	135.580
	12.8 M	14.820	13.183	13.244	13.286	13.033	12.962	13.413	12.919	13.473	13.559	133.892

Standard Buffer

1 Column with dictionary of size 800K - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	0.559	0.556	0.570	0.563	0.553	0.574	0.575	0.586	0.531	0.548	5.615
	200 K	0.542	0.545	0.541	0.541	0.543	0.594	0.580	0.567	0.572	0.572	5.597
	400 K	0.560	0.545	0.584	0.582	0.565	0.549	0.541	0.583	0.560	0.562	5.631
	800 K	0.583	0.602	0.563	0.555	0.573	0.552	0.589	0.564	0.580	0.588	5.749
	1.6 M	0.583	0.592	0.561	0.583	0.570	0.558	0.581	0.593	0.584	0.605	5.810
	3.2 M	0.599	0.596	0.544	0.567	0.556	0.572	0.543	0.568	0.618	0.543	5.706
	6.4 M	0.653	0.562	0.575	0.585	0.550	0.535	0.593	0.574	0.569	0.557	5.753
	12.8 M	0.682	0.547	0.580	0.563	0.588	0.594	0.580	0.573	0.575	0.575	5.857

1 Column with dictionary of size 800K - Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	27.661	25.665	26.783	26.474	26.176	26.366	25.591	27.896	25.666	25.762	264.040
	200 K	31.513	29.042	27.995	28.009	28.044	28.089	28.449	32.413	30.122	27.959	291.635
	400 K	30.716	29.925	29.953	30.039	31.582	30.956	30.642	32.010	30.137	29.829	305.789
	800 K	34.742	33.442	32.813	32.245	32.245	32.136	32.131	32.328	32.423	33.964	328.469
	1.6 M	24.671	21.824	21.845	21.779	21.513	21.052	21.098	21.365	21.117	21.083	217.347
	3.2 M	16.478	15.764	15.706	15.695	15.766	15.995	15.654	15.845	16.708	15.637	159.248
	6.4 M	13.927	12.868	12.911	12.972	12.944	12.973	14.312	13.020	12.865	12.888	131.680
	12.8 M	12.783	11.525	11.535	11.860	11.607	11.507	11.499	11.663	11.490	11.516	116.985

1 Column with dictionary of size 800K - Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	28.220	26.221	27.353	27.037	26.729	26.940	26.166	28.482	26.197	26.310	269.655
	200 K	32.055	29.587	28.536	28.550	28.587	28.683	29.029	32.980	30.694	28.531	297.232
	400 K	31.276	30.470	30.537	30.621	32.147	31.505	31.183	32.593	30.697	30.391	311.420
	800 K	35.325	34.044	33.376	32.800	32.818	32.688	32.720	32.892	33.003	34.552	334.218
	1.6 M	25.254	22.416	22.406	22.362	22.083	21.610	21.679	21.958	21.701	21.688	223.157
	3.2 M	17.077	16.360	16.250	16.262	16.322	16.567	16.197	16.413	17.326	16.180	164.954
	6.4 M	14.580	13.430	13.486	13.557	13.494	13.508	14.905	13.594	13.434	13.445	137.433
	12.8 M	13.465	12.072	12.115	12.423	12.195	12.101	12.079	12.236	12.065	12.091	122.842

Sort Buffer

1 Column with dictionary of size 800K - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	0.804	0.817	0.823	0.827	0.790	0.833	0.809	0.782	0.828	0.817	8.130
	200 K	0.828	0.793	0.814	0.823	0.813	0.831	0.805	0.873	0.820	0.800	8.200
	400 K	0.773	0.796	0.809	0.837	0.827	0.784	0.804	0.807	0.831	0.849	8.117
	800 K	0.822	0.799	0.803	0.848	0.835	0.809	0.846	0.812	0.824	0.783	8.181
	1.6 M	0.806	0.816	0.814	0.815	0.804	0.778	0.821	0.841	0.794	0.792	8.081
	3.2 M	0.824	0.802	0.796	0.808	0.805	0.816	0.816	0.802	0.804	0.836	8.109
	6.4 M	0.901	0.804	0.821	0.841	0.813	0.804	0.806	0.777	0.812	0.798	8.177
	12.8 M	0.940	0.772	0.811	0.816	0.849	0.833	0.820	0.779	0.821	0.786	8.227

1 Column with dictionary of size 800K - Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	9.522	9.037	9.051	9.213	9.184	9.014	9.043	9.192	9.048	9.079	91.383
	200 K	9.524	9.044	9.321	9.197	9.121	9.212	9.007	10.035	9.033	9.010	92.504
	400 K	9.858	9.454	9.489	9.534	9.605	9.427	9.495	9.676	9.437	10.080	96.055
	800 K	9.833	9.501	9.686	9.898	9.579	9.776	9.524	9.644	9.496	9.476	96.413
	1.6 M	8.481	8.053	7.897	7.937	7.958	7.869	7.891	8.077	7.857	7.858	79.878
	3.2 M	7.865	6.875	6.883	6.901	6.957	6.949	7.132	6.992	6.856	6.824	70.234
	6.4 M	10.059	6.844	6.873	6.882	7.271	6.813	6.840	7.157	6.821	6.810	72.370
	12.8 M	10.367	6.701	6.789	7.744	6.787	6.663	6.658	6.835	6.658	6.724	71.926

1 Column with dictionary of size 800K - Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	10.326	9.854	9.874	10.040	9.974	9.847	9.852	9.974	9.876	9.896	99.513
	200 K	10.352	9.837	10.135	10.020	9.934	10.043	9.812	10.908	9.853	9.810	100.704
	400 K	10.631	10.250	10.298	10.371	10.432	10.211	10.299	10.483	10.268	10.929	104.172
	800 K	10.655	10.300	10.489	10.746	10.414	10.585	10.370	10.456	10.320	10.259	104.594
	1.6 M	9.287	8.869	8.711	8.752	8.762	8.647	8.712	8.918	8.651	8.650	87.959
	3.2 M	8.689	7.677	7.679	7.709	7.762	7.765	7.948	7.794	7.660	7.660	78.343
	6.4 M	10.960	7.648	7.694	7.723	8.084	7.617	7.646	7.934	7.633	7.608	80.547
	12.8 M	11.307	7.473	7.600	8.560	7.636	7.496	7.478	7.614	7.479	7.510	80.153

Map Buffer

1 Column with dictionary of size 800K - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	0.576	0.534	0.552	0.562	0.560	0.538	0.528	0.555	0.565	0.558	5.528
	200 K	0.550	0.585	0.558	0.521	0.558	0.543	0.553	0.563	0.593	0.547	5.571
	400 K	0.576	0.538	0.602	0.582	0.557	0.569	0.565	0.550	0.573	0.585	5.697
	800 K	0.584	0.549	0.557	0.539	0.569	0.586	0.570	0.577	0.584	0.575	5.690
	1.6 M	0.596	0.555	0.593	0.568	0.548	0.560	0.574	0.569	0.566	0.570	5.699
	3.2 M	0.566	0.568	0.595	0.564	0.589	0.578	0.569	0.576	0.591	0.582	5.778
	6.4 M	0.613	0.564	0.558	0.563	0.584	0.565	0.546	0.567	0.573	0.584	5.717
	12.8 M	0.669	0.591	0.573	0.573	0.564	0.562	0.603	0.553	0.574	0.592	5.854

1 Column with dictionary of size 800K - Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	26.892	27.165	26.686	27.287	26.775	25.955	26.183	27.853	25.991	25.985	266.772
	200 K	29.182	28.612	28.647	28.368	29.439	29.233	29.365	30.335	28.371	28.597	290.149
	400 K	31.379	32.076	31.200	31.303	30.939	32.057	31.981	31.909	32.740	31.445	317.029
	800 K	34.064	32.962	33.195	34.150	33.326	34.088	32.708	32.887	35.483	32.687	335.550
	1.6 M	22.324	21.622	21.588	22.308	21.654	21.519	21.693	21.537	21.956	22.140	218.341
	3.2 M	16.706	16.909	15.967	15.969	17.036	16.020	15.876	15.890	16.152	15.908	162.433
	6.4 M	14.576	13.515	13.081	13.259	13.176	13.567	13.060	13.290	13.085	13.129	133.738
	12.8 M	12.419	12.547	11.750	11.759	11.761	11.722	11.853	11.907	11.659	11.705	119.082

1 Column with dictionary of size 800K - Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	27.468	27.699	27.238	27.849	27.335	26.493	26.711	28.408	26.556	26.543	272.300
	200 K	29.732	29.197	29.205	28.889	29.997	29.776	29.918	30.898	28.964	29.144	295.720
	400 K	31.955	32.614	31.802	31.885	31.496	32.626	32.546	32.459	33.313	32.030	322.726
	800 K	34.648	33.511	33.752	34.689	33.895	34.674	33.278	33.464	36.067	33.262	341.240
	1.6 M	22.920	22.177	22.181	22.876	22.202	22.079	22.267	22.106	22.522	22.710	224.040
	3.2 M	17.272	17.477	16.562	16.533	17.625	16.598	16.445	16.466	16.743	16.490	168.211
	6.4 M	15.189	14.079	13.639	13.822	13.760	14.132	13.606	13.857	13.658	13.713	139.455
	12.8 M	13.088	13.138	12.323	12.332	12.325	12.284	12.456	12.460	12.233	12.297	124.936

1,600,000 values in dictionary

No Buffer

1 Column with dictionary of size 1.6M - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	0 K	16.227	14.599	14.601	14.676	14.687	14.587	14.592	14.738	14.591	14.550	147.848

1 Column with dictionary of size 1.6M - Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	0 K	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

1 Column with dictionary of size 1.6M - Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	0 K	16.227	14.599	14.601	14.676	14.687	14.587	14.592	14.738	14.591	14.550	147.848

Index Buffer

1 Column with dictionary of size 1.6M - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	17.037	16.366	15.161	15.151	15.097	15.094	15.125	15.158	15.156	15.094	154.439
	200 K	17.701	15.114	15.471	15.447	15.134	15.750	15.750	19.557	15.128	15.079	160.131
	400 K	16.606	15.631	15.507	15.014	15.018	14.992	15.036	15.006	15.018	14.991	152.819
	800 K	17.446	16.039	15.034	15.638	15.043	15.040	15.081	15.694	18.518	15.072	158.605
	1.6 M	17.587	15.483	14.996	15.610	14.991	15.037	15.055	14.994	15.061	15.020	153.834
	3.2 M	16.624	14.993	15.063	15.041	17.359	15.688	14.945	17.507	19.064	16.889	163.173
	6.4 M	17.516	15.801	15.980	15.650	15.004	15.015	15.062	15.058	15.009	14.995	155.090
	12.8 M	16.768	15.067	15.692	18.064	15.248	15.692	15.150	15.178	17.152	15.100	159.111

1 Column with dictionary of size 1.6M - Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	0.087	0.062	0.078	0.093	0.139	0.038	0.186	0.027	0.028	0.030	0.768
	200 K	0.078	0.098	0.092	0.028	0.119	0.043	0.172	0.021	0.025	0.250	0.926
	400 K	0.075	0.087	0.085	0.113	0.032	0.161	0.026	0.028	0.215	0.026	0.848
	800 K	0.090	0.062	0.079	0.106	0.157	0.030	0.028	0.206	0.030	0.030	0.818
	1.6 M	0.079	0.103	0.100	0.028	0.142	0.031	0.186	0.030	0.028	0.265	0.992
	3.2 M	0.066	0.092	0.094	0.131	0.034	0.213	0.031	0.034	0.297	0.034	1.026
	6.4 M	0.042	0.095	0.094	0.100	0.030	0.167	0.030	0.242	0.030	0.030	0.860
	12.8 M	0.029	0.056	0.082	0.107	0.133	0.030	0.177	0.029	0.029	0.030	0.702

1 Column with dictionary of size 1.6M - Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	17.124	16.428	15.239	15.244	15.236	15.132	15.311	15.185	15.184	15.124	155.207
	200 K	17.779	15.212	15.563	15.475	15.253	15.793	15.922	19.578	15.153	15.329	161.057
	400 K	16.681	15.718	15.592	15.127	15.050	15.153	15.062	15.034	15.233	15.017	153.667
	800 K	17.536	16.101	15.113	15.744	15.200	15.070	15.109	15.900	18.548	15.102	159.423
	1.6 M	17.666	15.586	15.096	15.638	15.133	15.068	15.241	15.024	15.089	15.285	154.826
	3.2 M	16.690	15.085	15.157	15.172	17.393	15.901	14.976	17.541	19.361	16.923	164.199
	6.4 M	17.558	15.896	16.074	15.750	15.034	15.182	15.092	15.300	15.039	15.025	155.950
	12.8 M	16.797	15.123	15.774	18.171	15.381	15.722	15.327	15.207	17.181	15.130	159.813

Standard Buffer

1 Column with dictionary of size 1.6M - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	0.565	0.568	0.536	0.568	0.551	0.566	0.552	0.550	0.553	0.566	5.575
	200 K	0.590	0.550	0.582	0.565	0.568	0.601	0.563	0.596	0.557	0.560	5.732
	400 K	0.530	0.572	0.551	0.581	0.560	0.561	0.572	0.571	0.571	0.556	5.625
	800 K	0.562	0.563	0.522	0.561	0.559	0.590	0.586	0.581	0.592	0.586	5.702
	1.6 M	0.576	0.552	0.564	0.573	0.563	0.604	0.556	0.573	0.573	0.571	5.705
	3.2 M	0.611	0.576	0.599	0.586	0.571	0.572	0.574	0.567	0.556	0.558	5.770
	6.4 M	0.615	0.565	0.593	0.585	0.581	0.563	0.576	0.554	0.566	0.566	5.764
	12.8 M	0.673	0.603	0.552	0.599	0.571	0.598	0.572	0.581	0.597	0.581	5.927

1 Column with dictionary of size 1.6M - Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	30.954	27.942	27.914	27.954	28.004	27.884	27.977	29.446	27.931	28.222	284.228
	200 K	32.273	30.925	30.906	31.028	30.877	30.247	30.758	33.476	32.490	30.260	313.240
	400 K	34.974	32.182	32.236	32.854	32.276	32.159	33.425	32.959	32.924	32.749	328.738
	800 K	36.262	34.450	35.522	34.496	35.460	36.246	37.507	34.649	34.413	35.518	354.523
	1.6 M	38.357	37.536	37.937	38.040	36.590	36.543	36.504	36.672	37.940	36.492	372.611
	3.2 M	27.898	24.139	25.312	26.376	24.193	24.311	26.304	24.292	24.105	24.189	251.119
	6.4 M	19.833	21.093	19.249	17.993	18.282	17.896	17.906	18.094	17.929	17.928	186.203
	12.8 M	16.838	14.956	14.975	16.299	16.682	17.112	14.893	18.768	16.188	14.899	161.610

1 Column with dictionary of size 1.6M - Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	31.519	28.510	28.450	28.522	28.555	28.450	28.529	29.996	28.484	28.788	289.803
	200 K	32.863	31.475	31.488	31.593	31.445	30.848	31.321	34.072	33.047	30.820	318.972
	400 K	35.504	32.754	32.787	33.435	32.836	32.720	33.997	33.530	33.495	33.305	334.363
	800 K	36.824	35.013	36.044	35.057	36.019	36.836	38.093	35.230	35.005	36.104	360.225
	1.6 M	38.933	38.088	38.501	38.613	37.153	37.147	37.060	37.245	38.513	37.063	378.316
	3.2 M	28.509	24.715	25.911	26.962	24.764	24.883	26.878	24.859	24.661	24.747	256.889
	6.4 M	20.448	21.658	19.842	18.578	18.863	18.459	18.482	18.648	18.495	18.494	191.967
	12.8 M	17.511	15.559	15.527	16.898	17.253	17.710	15.465	19.349	16.785	15.480	167.537

Sort Buffer

1 Column with dictionary of size 1.6M - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	0.797	0.792	0.818	0.813	0.797	0.771	0.779	0.795	0.821	0.806	7.989
	200 K	0.797	0.781	0.806	0.791	0.798	0.829	0.815	0.787	0.822	0.836	8.062
	400 K	0.817	0.773	0.829	0.815	0.799	0.793	0.825	0.828	0.791	0.818	8.088
	800 K	0.831	0.837	0.804	0.798	0.827	0.811	0.800	0.808	0.811	0.829	8.156
	1.6 M	0.828	0.836	0.831	0.821	0.835	0.808	0.814	0.852	0.819	0.844	8.288
	3.2 M	0.839	0.803	0.803	0.816	0.848	0.799	0.826	0.814	0.790	0.823	8.161
	6.4 M	0.861	0.814	0.819	0.824	0.815	0.808	0.844	0.852	0.801	0.817	8.255
	12.8 M	0.955	0.843	0.834	0.808	0.793	0.823	0.794	0.813	0.808	0.799	8.270

1 Column with dictionary of size 1.6M - Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	12.081	11.131	11.156	11.187	11.219	11.100	11.106	11.305	11.628	11.113	113.026
	200 K	10.885	9.728	9.773	9.798	9.831	9.739	9.694	9.897	9.748	9.936	99.029
	400 K	10.946	10.037	9.984	9.969	9.990	9.817	9.811	10.628	9.842	9.843	100.867
	800 K	10.545	9.729	9.742	9.770	9.906	9.901	9.688	9.880	9.739	9.755	98.655
	1.6 M	11.162	10.354	10.976	10.409	10.443	10.346	10.332	10.521	10.360	10.330	105.233
	3.2 M	9.152	8.063	8.131	8.155	8.531	8.051	8.038	8.227	8.288	8.044	82.680
	6.4 M	10.002	7.426	7.433	7.581	7.496	7.385	7.402	7.552	7.387	7.394	77.058
	12.8 M	10.817	6.946	6.984	7.039	7.052	6.908	6.917	7.106	6.928	6.945	73.642

1 Column with dictionary of size 1.6M - Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	12.878	11.923	11.974	12.000	12.016	11.871	11.885	12.100	12.449	11.919	121.015
	200 K	11.682	10.509	10.579	10.589	10.629	10.568	10.509	10.684	10.570	10.772	107.091
	400 K	11.763	10.810	10.813	10.784	10.789	10.610	10.636	11.456	10.633	10.661	108.955
	800 K	11.376	10.566	10.546	10.568	10.733	10.712	10.488	10.688	10.550	10.584	106.811
	1.6 M	11.990	11.190	11.807	11.230	11.278	11.154	11.146	11.373	11.179	11.174	113.521
	3.2 M	9.991	8.866	8.934	8.971	9.379	8.850	8.864	9.041	9.078	8.867	90.841
	6.4 M	10.863	8.240	8.252	8.405	8.311	8.193	8.246	8.404	8.188	8.211	85.313
	12.8 M	11.772	7.789	7.818	7.847	7.845	7.731	7.711	7.919	7.736	7.744	81.912

Map Buffer

1 Column with dictionary of size 1.6M - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	0.561	0.576	0.566	0.550	0.554	0.557	0.533	0.559	0.577	0.568	5.601
	200 K	0.557	0.589	0.548	0.549	0.566	0.555	0.536	0.550	0.598	0.555	5.603
	400 K	0.580	0.576	0.532	0.606	0.575	0.553	0.557	0.561	0.573	0.579	5.692
	800 K	0.560	0.616	0.562	0.598	0.590	0.579	0.566	0.562	0.563	0.560	5.756
	1.6 M	0.602	0.566	0.615	0.560	0.558	0.585	0.573	0.555	0.543	0.568	5.725
	3.2 M	0.587	0.554	0.586	0.555	0.560	0.554	0.576	0.561	0.554	0.574	5.661
	6.4 M	0.645	0.583	0.566	0.563	0.608	0.584	0.599	0.573	0.580	0.557	5.858
	12.8 M	0.672	0.576	0.576	0.577	0.569	0.596	0.578	0.567	0.543	0.547	5.801

1 Column with dictionary of size 1.6M - Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	30.963	28.386	30.194	28.382	28.843	28.989	28.713	34.310	29.546	28.991	297.317
	200 K	32.805	32.991	30.829	32.228	32.079	30.662	30.966	33.486	34.387	33.145	323.578
	400 K	34.649	34.090	32.851	35.065	34.437	33.646	33.399	32.807	33.672	35.126	339.742
	800 K	36.857	37.522	35.273	35.767	35.121	38.440	35.152	35.227	35.037	36.792	361.188
	1.6 M	42.717	38.517	38.556	37.015	38.077	39.072	37.196	38.011	37.685	38.619	385.465
	3.2 M	26.352	24.595	25.951	24.663	24.505	28.351	27.898	24.668	24.794	26.688	258.465
	6.4 M	20.085	20.019	18.306	19.060	18.285	18.440	18.268	18.490	18.278	18.377	187.608
	12.8 M	16.866	15.137	15.158	15.226	15.374	15.117	16.478	15.110	15.115	15.092	154.673

1 Column with dictionary of size 1.6M - Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	31.524	28.962	30.760	28.932	29.397	29.546	29.246	34.869	30.123	29.559	302.918
	200 K	33.362	33.580	31.377	32.777	32.645	31.217	31.502	34.036	34.985	33.700	329.181
	400 K	35.229	34.666	33.383	35.671	35.012	34.199	33.956	33.368	34.245	35.705	345.434
	800 K	37.417	38.138	35.835	36.365	35.711	39.019	35.718	35.789	35.600	37.352	366.944
	1.6 M	43.319	39.083	39.171	37.575	38.635	39.657	37.769	38.566	38.228	39.187	391.190
	3.2 M	26.939	25.149	26.537	25.218	25.065	28.905	28.474	25.229	25.348	27.262	264.126
	6.4 M	20.730	20.602	18.872	19.623	18.893	19.024	18.867	19.063	18.858	18.934	193.466
	12.8 M	17.538	15.713	15.734	15.803	15.943	15.713	17.056	15.677	15.658	15.639	160.474

3,200,000 values in dictionary

No Buffer

1 Column with dictionary of size 3.2M - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	0 K	20.379	16.940	16.542	16.725	16.769	16.689	16.687	16.896	17.394	19.175	174.196

1 Column with dictionary of size 3.2M - Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	0 K	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

1 Column with dictionary of size 3.2M - Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	0 K	20.379	16.940	16.542	16.725	16.769	16.689	16.687	16.896	17.394	19.175	174.196

Index Buffer

1 Column with dictionary of size 3.2M - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	20.745	17.211	18.652	18.613	17.223	17.627	17.797	20.877	22.686	17.233	188.664
	200 K	21.097	18.341	17.249	17.224	17.269	17.232	17.218	17.181	17.230	17.217	177.258
	400 K	20.715	17.512	17.254	17.244	17.204	18.463	17.232	17.673	20.872	19.216	183.385
	800 K	20.887	17.495	17.141	17.147	19.066	17.556	17.194	17.176	20.021	22.421	186.104
	1.6 M	20.740	17.627	17.181	17.587	17.570	17.907	17.168	17.177	20.793	22.723	186.473
	3.2 M	21.811	17.175	17.873	17.119	17.190	17.181	17.217	17.153	17.236	17.174	177.129
	6.4 M	20.737	17.155	17.488	17.402	18.056	17.203	17.227	21.979	23.411	23.349	194.007
	12.8 M	20.750	17.134	17.140	18.062	17.966	17.203	17.714	17.968	18.830	23.249	186.016

1 Column with dictionary of size 3.2M - Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	0.092	0.058	0.070	0.109	0.143	0.038	0.185	0.037	0.041	0.030	0.803
	200 K	0.087	0.101	0.086	0.027	0.122	0.026	0.182	0.029	0.029	0.278	0.967
	400 K	0.073	0.092	0.085	0.115	0.031	0.193	0.029	0.030	0.233	0.035	0.916
	800 K	0.090	0.064	0.081	0.106	0.157	0.030	0.028	0.204	0.030	0.032	0.822
	1.6 M	0.076	0.106	0.102	0.030	0.139	0.029	0.189	0.030	0.034	0.261	0.996
	3.2 M	0.068	0.089	0.118	0.126	0.029	0.179	0.028	0.030	0.266	0.029	0.962
	6.4 M	0.043	0.093	0.080	0.128	0.029	0.168	0.029	0.262	0.036	0.038	0.906
	12.8 M	0.030	0.056	0.082	0.107	0.137	0.030	0.197	0.029	0.040	0.039	0.747

1 Column with dictionary of size 3.2M - Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	20.837	17.269	18.722	18.722	17.366	17.665	17.982	20.914	22.727	17.263	189.467
	200 K	21.184	18.442	17.335	17.251	17.391	17.258	17.400	17.210	17.259	17.495	178.225
	400 K	20.788	17.604	17.339	17.359	17.235	18.656	17.261	17.703	21.105	19.251	184.301
	800 K	20.977	17.559	17.222	17.253	19.223	17.586	17.222	17.380	20.051	22.453	186.926
	1.6 M	20.816	17.733	17.283	17.617	17.709	17.936	17.357	17.207	20.827	22.984	187.469
	3.2 M	21.879	17.264	17.991	17.245	17.219	17.360	17.245	17.183	17.502	17.203	178.091
	6.4 M	20.780	17.248	17.568	17.530	18.085	17.371	17.256	22.241	23.447	23.387	194.913
	12.8 M	20.780	17.190	17.222	18.169	18.103	17.233	17.911	17.997	18.870	23.288	186.763

Standard Buffer

1 Column with dictionary of size 3.2M - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	0.566	0.581	0.584	0.587	0.571	0.589	0.546	0.576	0.567	0.544	5.711
	200 K	0.591	0.565	0.572	0.562	0.577	0.546	0.551	0.550	0.556	0.580	5.650
	400 K	0.567	0.591	0.536	0.546	0.576	0.561	0.561	0.544	0.575	0.567	5.624
	800 K	0.574	0.597	0.596	0.594	0.577	0.531	0.573	0.571	0.561	0.590	5.764
	1.6 M	0.583	0.560	0.563	0.587	0.559	0.567	0.581	0.525	0.596	0.548	5.669
	3.2 M	0.574	0.593	0.568	0.555	0.567	0.565	0.576	0.562	0.555	0.535	5.650
	6.4 M	0.627	0.570	0.579	0.545	0.552	0.552	0.585	0.572	0.584	0.556	5.722
	12.8 M	0.682	0.555	0.578	0.576	0.564	0.558	0.580	0.552	0.579	0.560	5.784

1 Column with dictionary of size 3.2M - Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	36.632	30.171	31.374	30.871	31.403	30.470	30.769	30.371	32.343	30.197	314.601
	200 K	38.838	34.993	33.617	34.828	33.081	33.399	32.776	34.896	32.547	32.542	341.517
	400 K	38.396	34.817	35.190	35.120	35.754	35.018	34.577	38.136	34.388	35.457	356.853
	800 K	40.632	42.405	40.306	37.135	37.837	36.834	37.397	38.928	37.389	36.792	385.655
	1.6 M	45.895	38.824	40.011	39.032	38.955	38.797	38.835	39.092	41.441	38.842	399.724
	3.2 M	45.050	41.799	41.169	41.224	41.240	44.452	44.010	41.871	41.144	41.526	423.485
	6.4 M	32.565	31.988	27.430	27.654	27.511	27.445	27.438	27.621	27.433	30.605	287.690
	12.8 M	24.550	20.565	20.596	20.897	20.680	20.566	20.597	20.917	20.598	20.571	210.537

1 Column with dictionary of size 3.2M - Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	37.198	30.752	31.958	31.458	31.974	31.059	31.315	30.947	32.910	30.741	320.312
	200 K	39.429	35.558	34.189	35.390	33.658	33.945	33.327	35.446	33.103	33.122	347.167
	400 K	38.963	35.408	35.726	35.666	36.330	35.579	35.138	38.680	34.963	36.024	362.477
	800 K	41.206	43.002	40.902	37.729	38.414	37.365	37.970	39.499	37.950	37.382	391.419
	1.6 M	46.478	39.384	40.574	39.619	39.514	39.364	39.416	39.617	42.037	39.390	405.393
	3.2 M	45.624	42.392	41.737	41.779	41.807	45.017	44.586	42.433	41.699	42.061	429.135
	6.4 M	33.192	32.558	28.009	28.199	28.063	27.997	28.023	28.193	28.017	31.161	293.412
	12.8 M	25.232	21.120	21.174	21.473	21.244	21.124	21.177	21.469	21.177	21.131	216.321

Sort Buffer

1 Column with dictionary of size 3.2M - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	0.792	0.779	0.793	0.802	0.824	0.804	0.777	0.830	0.826	0.821	8.048
	200 K	0.787	0.805	0.807	0.808	0.827	0.838	0.813	0.817	0.814	0.824	8.140
	400 K	0.816	0.811	0.849	0.816	0.817	0.811	0.821	0.843	0.808	0.803	8.195
	800 K	0.823	0.804	0.807	0.815	0.803	0.796	0.816	0.809	0.825	0.795	8.093
	1.6 M	0.798	0.813	0.812	0.812	0.813	0.803	0.788	0.808	0.814	0.792	8.053
	3.2 M	0.857	0.767	0.789	0.813	0.799	0.807	0.822	0.775	0.821	0.806	8.056
	6.4 M	0.880	0.781	0.791	0.807	0.822	0.817	0.783	0.832	0.810	0.798	8.121
	12.8 M	0.899	0.830	0.847	0.817	0.832	0.806	0.811	0.830	0.838	0.811	8.321

1 Column with dictionary of size 3.2M - Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	14.621	12.658	13.229	12.697	12.965	12.614	12.621	12.806	12.625	12.654	129.490
	200 K	13.529	11.741	11.757	11.780	12.113	11.699	11.708	12.095	11.741	11.727	119.890
	400 K	12.373	10.468	10.604	10.493	10.534	10.531	10.419	11.206	10.426	10.459	107.513
	800 K	11.865	10.059	10.077	10.095	10.268	10.029	10.035	10.227	10.060	10.045	102.760
	1.6 M	12.130	10.729	10.820	10.641	10.727	10.543	10.554	10.729	10.527	10.542	107.942
	3.2 M	11.854	10.618	10.634	10.940	10.693	10.565	10.634	10.728	10.614	10.584	107.864
	6.4 M	10.825	8.684	8.686	9.335	8.758	8.632	8.650	8.827	8.641	8.656	89.694
	12.8 M	11.778	7.539	7.568	7.602	7.643	7.504	7.517	8.050	7.508	7.496	80.205

1 Column with dictionary of size 3.2M - Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	15.413	13.437	14.022	13.499	13.789	13.418	13.398	13.636	13.451	13.475	137.538
	200 K	14.316	12.546	12.564	12.588	12.940	12.537	12.521	12.912	12.555	12.551	128.030
	400 K	13.189	11.279	11.453	11.309	11.351	11.342	11.240	12.049	11.234	11.262	115.708
	800 K	12.688	10.863	10.884	10.910	11.071	10.825	10.851	11.036	10.885	10.840	110.853
	1.6 M	12.928	11.542	11.632	11.453	11.540	11.346	11.342	11.537	11.341	11.334	115.995
	3.2 M	12.711	11.385	11.423	11.753	11.492	11.372	11.456	11.503	11.435	11.390	115.920
	6.4 M	11.705	9.465	9.477	10.142	9.580	9.449	9.433	9.659	9.451	9.454	97.815
	12.8 M	12.677	8.369	8.415	8.419	8.475	8.310	8.328	8.880	8.346	8.307	88.526

Map Buffer

1 Column with dictionary of size 3.2M - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	0.561	0.527	0.573	0.549	0.566	0.549	0.553	0.569	0.533	0.526	5.506
	200 K	0.577	0.565	0.568	0.617	0.519	0.571	0.558	0.579	0.559	0.570	5.683
	400 K	0.522	0.553	0.556	0.590	0.558	0.555	0.567	0.548	0.545	0.566	5.560
	800 K	0.565	0.576	0.568	0.581	0.597	0.598	0.556	0.565	0.577	0.562	5.745
	1.6 M	0.574	0.586	0.551	0.578	0.565	0.571	0.560	0.592	0.580	0.580	5.737
	3.2 M	0.584	0.560	0.595	0.540	0.569	0.565	0.601	0.562	0.594	0.542	5.712
	6.4 M	0.599	0.603	0.567	0.577	0.543	0.553	0.561	0.574	0.539	0.595	5.711
	12.8 M	0.665	0.563	0.553	0.556	0.552	0.574	0.527	0.580	0.571	0.550	5.691

1 Column with dictionary of size 3.2M - Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	34.550	30.569	31.194	31.242	31.075	30.576	31.180	32.502	30.547	30.597	314.032
	200 K	38.214	33.922	33.134	39.805	33.188	33.111	33.268	34.862	33.079	35.184	347.767
	400 K	39.324	35.669	36.300	36.037	36.765	35.312	35.134	35.155	35.393	35.150	360.239
	800 K	41.185	39.621	38.207	37.560	41.678	37.614	37.187	37.416	37.246	39.779	387.493
	1.6 M	43.253	40.591	40.803	42.005	39.883	42.222	42.660	42.908	39.581	39.708	413.614
	3.2 M	46.556	45.217	47.070	43.341	41.790	44.661	41.644	41.634	41.912	41.617	435.442
	6.4 M	31.623	30.290	30.222	27.904	29.335	29.003	27.722	28.907	30.729	27.674	293.409
	12.8 M	24.658	20.829	20.831	20.867	20.924	20.774	20.995	20.843	21.142	20.842	212.705

1 Column with dictionary of size 3.2M - Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	35.111	31.096	31.767	31.791	31.641	31.125	31.733	33.071	31.080	31.123	319.538
	200 K	38.791	34.487	33.702	40.422	33.707	33.682	33.826	35.441	33.638	35.754	353.450
	400 K	39.846	36.222	36.856	36.627	37.323	35.867	35.701	35.703	35.938	35.716	365.799
	800 K	41.750	40.197	38.775	38.141	42.275	38.212	37.743	37.981	37.823	40.341	393.238
	1.6 M	43.827	41.177	41.354	42.583	40.448	42.793	43.220	43.500	40.161	40.288	419.351
	3.2 M	47.140	45.777	47.665	43.881	42.359	45.226	42.245	42.196	42.506	42.159	441.154
	6.4 M	32.222	30.893	30.789	28.481	29.878	29.556	28.283	29.481	31.268	28.269	299.120
	12.8 M	25.323	21.392	21.384	21.423	21.476	21.348	21.522	21.423	21.713	21.392	218.396

6,400,000 values in dictionary

No Buffer

1 Column with dictionary of size 6.4M - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	0 K	26.784	19.196	20.142	19.098	19.317	19.234	19.344	19.219	18.998	21.617	202.949

1 Column with dictionary of size 6.4M - Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	0 K	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

1 Column with dictionary of size 6.4M - Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	0 K	26.784	19.196	20.142	19.098	19.317	19.234	19.344	19.219	18.998	21.617	202.949

Index Buffer

1 Column with dictionary of size 6.4M - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	29.981	22.680	19.789	19.732	19.872	19.883	19.764	19.763	22.365	21.561	215.390
	200 K	27.498	22.417	20.961	20.075	19.672	20.382	20.234	22.504	21.107	23.334	218.184
	400 K	28.773	19.622	19.700	19.661	19.645	19.700	19.681	19.658	20.097	19.659	206.196
	800 K	27.450	19.743	20.381	20.205	20.147	20.822	19.730	20.069	21.987	19.709	210.243
	1.6 M	28.215	20.250	19.855	19.827	19.803	19.777	19.771	19.801	19.770	19.904	206.973
	3.2 M	27.219	19.728	20.695	22.369	19.682	20.163	19.713	21.500	20.304	19.682	211.055
	6.4 M	27.301	19.672	19.668	20.440	19.728	19.699	20.574	21.458	20.648	21.223	210.411
	12.8 M	29.334	20.194	21.065	20.694	19.711	19.732	19.688	19.636	25.376	21.135	216.565

1 Column with dictionary of size 6.4M - Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	0.097	0.066	0.069	0.101	0.142	0.023	0.206	0.033	0.041	0.037	0.815
	200 K	0.086	0.119	0.098	0.038	0.132	0.031	0.188	0.031	0.039	0.335	1.097
	400 K	0.080	0.095	0.088	0.115	0.027	0.168	0.029	0.037	0.229	0.035	0.903
	800 K	0.092	0.063	0.081	0.108	0.142	0.030	0.033	0.207	0.034	0.034	0.824
	1.6 M	0.078	0.109	0.098	0.030	0.129	0.030	0.190	0.030	0.030	0.296	1.020
	3.2 M	0.067	0.091	0.097	0.131	0.031	0.167	0.030	0.032	0.273	0.030	0.949
	6.4 M	0.042	0.092	0.079	0.124	0.030	0.171	0.029	0.245	0.033	0.030	0.875
	12.8 M	0.034	0.056	0.081	0.111	0.121	0.029	0.202	0.030	0.040	0.030	0.734

1 Column with dictionary of size 6.4M - Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	30.078	22.746	19.858	19.833	20.014	19.906	19.970	19.796	22.406	21.598	216.205
	200 K	27.584	22.536	21.059	20.113	19.804	20.413	20.422	22.535	21.146	23.669	219.281
	400 K	28.853	19.717	19.788	19.776	19.672	19.868	19.710	19.695	20.326	19.694	207.099
	800 K	27.542	19.806	20.462	20.313	20.289	20.852	19.763	20.276	22.021	19.743	211.067
	1.6 M	28.293	20.359	19.953	19.857	19.932	19.807	19.961	19.831	19.800	20.200	207.993
	3.2 M	27.286	19.819	20.792	22.500	19.713	20.330	19.743	21.532	20.577	19.712	212.004
	6.4 M	27.343	19.764	19.747	20.564	19.758	19.870	20.603	21.703	20.681	21.253	211.286
	12.8 M	29.368	20.250	21.146	20.805	19.832	19.761	19.890	19.666	25.416	21.165	217.299

Standard Buffer

1 Column with dictionary of size 6.4M - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	0.570	0.575	0.539	0.544	0.555	0.559	0.566	0.570	0.563	0.573	5.614
	200 K	0.555	0.539	0.552	0.565	0.545	0.577	0.567	0.555	0.547	0.561	5.563
	400 K	0.577	0.565	0.571	0.568	0.543	0.553	0.578	0.562	0.550	0.573	5.640
	800 K	0.606	0.565	0.578	0.583	0.598	0.569	0.559	0.599	0.575	0.613	5.845
	1.6 M	0.584	0.571	0.585	0.568	0.579	0.559	0.571	0.576	0.550	0.558	5.701
	3.2 M	0.570	0.575	0.553	0.537	0.560	0.571	0.553	0.584	0.555	0.571	5.629
	6.4 M	0.633	0.564	0.549	0.563	0.563	0.569	0.576	0.609	0.578	0.592	5.796
	12.8 M	0.685	0.573	0.601	0.595	0.586	0.565	0.569	0.549	0.575	0.553	5.851

1 Column with dictionary of size 6.4M - Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	41.069	36.212	33.726	35.768	36.797	32.951	32.767	32.998	32.818	34.320	349.426
	200 K	44.279	35.046	35.115	35.046	35.085	35.904	36.326	36.283	34.991	35.024	363.099
	400 K	45.552	38.210	38.020	38.304	37.983	38.887	37.176	37.391	37.195	39.285	388.003
	800 K	51.512	39.344	40.228	39.403	39.942	40.116	39.270	42.011	43.816	43.131	418.773
	1.6 M	50.157	42.839	43.516	41.451	41.504	41.450	43.427	41.798	41.510	41.515	429.167
	3.2 M	52.267	45.685	45.824	43.921	45.917	43.897	44.855	44.097	43.935	47.008	457.406
	6.4 M	55.076	49.992	48.694	53.423	46.833	46.898	46.699	50.049	49.015	48.593	495.272
	12.8 M	39.475	31.497	35.252	31.398	31.439	31.341	31.319	31.501	31.623	31.267	326.112

1 Column with dictionary of size 6.4M - Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	41.639	36.787	34.265	36.312	37.352	33.510	33.333	33.568	33.381	34.893	355.040
	200 K	44.834	35.585	35.667	35.611	35.630	36.481	36.893	36.838	35.538	35.585	368.662
	400 K	46.129	38.775	38.591	38.872	38.526	39.440	37.754	37.953	37.745	39.858	393.643
	800 K	52.118	39.909	40.806	39.986	40.540	40.685	39.829	42.610	44.391	43.744	424.618
	1.6 M	50.741	43.410	44.101	42.019	42.083	42.009	43.998	42.374	42.060	42.073	434.868
	3.2 M	52.837	46.260	46.377	44.458	46.477	44.468	45.408	44.681	44.490	47.579	463.035
	6.4 M	55.709	50.556	49.243	53.986	47.396	47.467	47.275	50.658	49.593	49.185	501.068
	12.8 M	40.160	32.070	35.853	31.993	32.025	31.906	31.888	32.050	32.198	31.820	331.963

Sort Buffer

1 Column with dictionary of size 6.4M - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	0.819	0.811	0.783	0.801	0.823	0.789	0.809	0.804	0.800	0.839	8.078
	200 K	0.837	0.791	0.818	0.794	0.783	0.837	0.804	0.815	0.822	0.807	8.108
	400 K	0.821	0.835	0.811	0.774	0.803	0.809	0.815	0.820	0.818	0.793	8.099
	800 K	0.794	0.811	0.787	0.801	0.836	0.798	0.820	0.807	0.808	0.802	8.064
	1.6 M	0.786	0.827	0.804	0.791	0.799	0.830	0.815	0.807	0.830	0.803	8.092
	3.2 M	0.858	0.831	0.798	0.793	0.811	0.781	0.846	0.844	0.982	0.900	8.444
	6.4 M	0.879	0.849	0.774	0.824	0.831	0.823	0.806	0.802	0.815	0.826	8.229
	12.8 M	0.917	0.844	0.850	0.820	0.818	0.830	0.824	0.829	0.792	0.828	8.352

1 Column with dictionary of size 6.4M - Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	18.868	14.335	14.353	14.907	14.655	14.293	14.333	14.448	14.315	15.263	149.770
	200 K	17.520	13.910	14.130	13.707	13.525	13.419	13.419	13.588	13.444	13.439	140.101
	400 K	16.522	13.580	12.844	13.543	12.706	12.616	12.615	12.814	12.664	13.030	132.934
	800 K	14.818	10.941	10.986	10.979	11.017	10.984	10.973	11.093	10.920	10.919	113.630
	1.6 M	14.618	10.947	10.945	10.962	11.009	11.086	10.919	11.356	12.112	11.375	115.329
	3.2 M	14.101	10.854	11.081	10.965	10.991	10.809	10.825	11.390	15.126	12.986	119.128
	6.4 M	13.934	13.267	11.370	11.389	11.471	11.833	11.379	11.501	11.366	12.152	119.662
	12.8 M	13.317	8.864	8.890	8.943	8.922	8.837	8.831	9.301	9.866	8.876	94.647

1 Column with dictionary of size 6.4M - Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	19.687	15.146	15.136	15.708	15.478	15.082	15.142	15.252	15.115	16.102	157.848
	200 K	18.357	14.701	14.948	14.501	14.308	14.256	14.223	14.403	14.266	14.246	148.209
	400 K	17.343	14.415	13.655	14.317	13.509	13.425	13.430	13.634	13.482	13.823	141.033
	800 K	15.612	11.752	11.773	11.780	11.853	11.782	11.793	11.900	11.728	11.721	121.694
	1.6 M	15.404	11.774	11.749	11.753	11.808	11.916	11.734	12.163	12.942	12.178	123.421
	3.2 M	14.959	11.685	11.879	11.758	11.802	11.590	11.671	12.234	16.108	13.886	127.572
	6.4 M	14.813	14.116	12.144	12.213	12.302	12.656	12.185	12.303	12.181	12.978	127.891
	12.8 M	14.234	9.708	9.740	9.763	9.740	9.667	9.655	10.130	10.658	9.704	102.999

Map Buffer

1 Column with dictionary of size 6.4M - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	0.546	0.549	0.565	0.559	0.569	0.564	0.575	0.590	0.547	0.575	5.639
	200 K	0.521	0.542	0.557	0.528	0.561	0.578	0.557	0.513	0.575	0.575	5.507
	400 K	0.579	0.535	0.575	0.563	0.577	0.527	0.558	0.521	0.576	0.580	5.591
	800 K	0.573	0.590	0.573	0.560	0.552	0.549	0.588	0.588	0.562	0.556	5.691
	1.6 M	0.566	0.582	0.545	0.580	0.577	0.570	0.572	0.577	0.554	0.564	5.687
	3.2 M	0.559	0.571	0.563	0.554	0.570	0.568	0.538	0.578	0.613	0.559	5.673
	6.4 M	0.622	0.582	0.556	0.586	0.565	0.595	0.581	0.582	0.555	0.553	5.777
	12.8 M	0.679	0.564	0.566	0.566	0.578	0.557	0.562	0.597	0.587	0.563	5.819

1 Column with dictionary of size 6.4M - Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	41.426	33.251	34.582	35.314	33.369	33.276	35.494	42.220	34.486	33.236	356.654
	200 K	43.586	35.336	37.248	35.468	35.435	35.993	35.472	37.763	35.384	35.610	367.295
	400 K	46.206	38.728	38.517	38.893	38.446	38.211	38.844	37.627	37.972	38.456	391.900
	800 K	48.625	39.942	39.946	41.331	41.016	42.012	39.916	42.273	41.508	39.886	416.455
	1.6 M	52.865	43.245	42.012	42.087	44.517	44.586	43.624	41.947	42.843	42.318	440.044
	3.2 M	52.862	46.456	48.547	46.152	48.348	47.628	44.328	46.357	46.695	44.387	471.760
	6.4 M	55.234	48.503	51.688	48.586	47.579	50.252	47.820	47.959	49.638	47.303	494.562
	12.8 M	39.627	32.081	31.815	32.115	33.396	32.970	31.800	31.617	31.602	31.645	328.668

1 Column with dictionary of size 6.4M - Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	41.972	33.800	35.147	35.873	33.938	33.840	36.069	42.810	35.033	33.811	362.293
	200 K	44.107	35.878	37.805	35.996	35.996	36.571	36.029	38.276	35.959	36.185	372.802
	400 K	46.785	39.263	39.092	39.456	39.023	38.738	39.402	38.148	38.548	39.036	397.491
	800 K	49.198	40.532	40.519	41.891	41.568	42.561	40.504	42.861	42.070	40.442	422.146
	1.6 M	53.431	43.827	42.557	42.667	45.094	45.156	44.196	42.524	43.397	42.882	445.731
	3.2 M	53.421	47.027	49.110	46.706	48.918	48.196	44.866	46.935	47.308	44.946	477.433
	6.4 M	55.856	49.085	52.244	49.172	48.144	50.847	48.401	48.541	50.193	47.856	500.339
	12.8 M	40.306	32.645	32.381	32.681	33.974	33.527	32.362	32.214	32.189	32.208	334.487

12,800,000 values in dictionary

No Buffer

1 Column with dictionary of size 12.8M - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	0 K	37.609	22.030	21.493	21.684	21.667	21.608	21.774	21.805	21.557	21.638	232.865

1 Column with dictionary of size 12.8M - Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	0 K	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

1 Column with dictionary of size 12.8M - Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	0 K	37.609	22.030	21.493	21.684	21.667	21.608	21.774	21.805	21.557	21.638	232.865

Index Buffer

1 Column with dictionary of size 12.8M - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	39.859	22.325	22.375	22.381	22.504	22.403	22.929	22.373	25.658	22.444	245.251
	200 K	38.074	24.025	22.429	22.910	22.374	22.350	22.750	23.300	22.363	22.421	242.996
	400 K	38.804	22.792	23.823	22.371	22.868	22.964	22.337	22.320	28.075	22.315	248.669
	800 K	38.040	22.661	23.930	22.331	22.276	23.483	22.348	22.908	25.222	22.328	245.527
	1.6 M	38.613	22.317	22.368	22.349	24.311	23.803	22.909	22.275	22.278	25.782	247.005
	3.2 M	38.433	22.333	23.382	22.897	22.302	22.341	22.337	22.835	23.782	26.640	247.282
	6.4 M	39.500	22.932	22.994	22.358	22.777	22.373	22.364	26.121	25.557	22.372	249.348
	12.8 M	38.940	22.285	23.180	23.002	22.197	22.363	22.381	22.265	22.242	22.277	241.132

1 Column with dictionary of size 12.8M - Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	0.090	0.060	0.081	0.113	0.133	0.032	0.204	0.032	0.032	0.036	0.813
	200 K	0.083	0.097	0.100	0.035	0.123	0.032	0.182	0.033	0.032	0.276	0.993
	400 K	0.076	0.092	0.097	0.112	0.026	0.163	0.028	0.030	0.226	0.029	0.879
	800 K	0.090	0.077	0.082	0.105	0.158	0.033	0.027	0.205	0.029	0.028	0.834
	1.6 M	0.079	0.103	0.099	0.030	0.146	0.028	0.194	0.029	0.030	0.268	1.006
	3.2 M	0.067	0.091	0.092	0.131	0.030	0.161	0.029	0.030	0.275	0.033	0.939
	6.4 M	0.043	0.093	0.081	0.108	0.030	0.149	0.029	0.294	0.031	0.030	0.888
	12.8 M	0.031	0.057	0.099	0.109	0.136	0.030	0.180	0.030	0.030	0.029	0.731

1 Column with dictionary of size 12.8M - Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	39.949	22.385	22.456	22.494	22.637	22.435	23.133	22.405	25.690	22.480	246.064
	200 K	38.157	24.122	22.529	22.945	22.497	22.382	22.932	23.333	22.395	22.697	243.989
	400 K	38.880	22.884	23.920	22.483	22.894	23.127	22.365	22.350	28.301	22.344	249.548
	800 K	38.130	22.738	24.012	22.436	22.434	23.516	22.375	23.113	25.251	22.356	246.361
	1.6 M	38.692	22.420	22.467	22.379	24.457	23.831	23.103	22.304	22.308	26.050	248.011
	3.2 M	38.500	22.424	23.474	23.028	22.332	22.502	22.366	22.865	24.057	26.673	248.221
	6.4 M	39.543	23.025	23.075	22.466	22.807	22.522	22.393	26.415	25.588	22.402	250.236
	12.8 M	38.971	22.342	23.279	23.111	22.333	22.393	22.561	22.295	22.272	22.306	241.863

Standard Buffer

1 Column with dictionary of size 12.8M - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	0.526	0.578	0.569	0.565	0.519	0.565	0.557	0.562	0.573	0.558	5.572
	200 K	0.557	0.563	0.516	0.542	0.532	0.579	0.551	0.552	0.548	0.535	5.475
	400 K	0.592	0.599	0.570	0.570	0.552	0.528	0.545	0.566	0.592	0.546	5.660
	800 K	0.595	0.551	0.553	0.552	0.570	0.558	0.570	0.583	0.557	0.588	5.677
	1.6 M	0.578	0.601	0.571	0.564	0.574	0.570	0.588	0.566	0.555	0.566	5.733
	3.2 M	0.602	0.573	0.569	0.591	0.557	0.552	0.549	0.569	0.568	0.575	5.705
	6.4 M	0.623	0.554	0.560	0.563	0.572	0.578	0.557	0.564	0.581	0.589	5.741
	12.8 M	0.673	0.552	0.542	0.552	0.582	0.557	0.576	0.581	0.543	0.564	5.722

1 Column with dictionary of size 12.8M - Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	52.832	36.343	36.125	36.149	37.291	36.709	36.123	36.368	37.531	36.076	381.547
	200 K	55.530	38.139	40.069	38.270	39.287	38.959	40.288	38.278	38.077	38.628	405.525
	400 K	58.179	40.638	41.380	41.885	41.702	42.906	40.376	40.894	40.515	42.011	430.486
	800 K	62.774	43.911	43.450	44.241	43.233	44.324	43.286	43.740	42.702	45.667	457.328
	1.6 M	62.460	45.202	46.375	44.842	47.924	46.493	46.537	45.108	44.832	44.803	474.576
	3.2 M	64.127	49.801	48.414	47.560	48.084	49.386	47.166	47.270	47.074	47.191	496.073
	6.4 M	67.081	50.283	50.380	50.372	55.034	50.096	54.365	50.254	50.061	50.049	527.975
	12.8 M	70.090	54.764	56.335	53.884	53.913	60.923	56.305	55.905	53.731	53.786	569.636

1 Column with dictionary of size 12.8M - Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	53.358	36.921	36.694	36.714	37.810	37.274	36.680	36.930	38.104	36.634	387.119
	200 K	56.087	38.702	40.585	38.812	39.819	39.538	40.839	38.830	38.625	39.163	411.000
	400 K	58.771	41.237	41.950	42.455	42.254	43.434	40.921	41.460	41.107	42.557	436.146
	800 K	63.369	44.462	44.003	44.793	43.803	44.882	43.856	44.323	43.259	46.255	463.005
	1.6 M	63.038	45.803	46.946	45.406	48.498	47.063	47.125	45.674	45.387	45.369	480.309
	3.2 M	64.729	50.374	48.983	48.151	48.641	49.938	47.715	47.839	47.642	47.766	501.778
	6.4 M	67.704	50.837	50.940	50.935	55.606	50.674	54.922	50.818	50.642	50.638	533.716
	12.8 M	70.763	55.316	56.877	54.436	54.495	61.480	56.881	56.486	54.274	54.350	575.358

Sort Buffer

1 Column with dictionary of size 12.8M - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	0.824	0.804	0.833	0.805	0.822	0.822	0.786	0.820	0.789	0.788	8.093
	200 K	0.810	0.781	0.796	0.796	0.809	0.787	0.808	0.809	0.804	0.808	8.008
	400 K	0.808	0.812	0.791	0.796	0.795	0.814	0.842	0.847	0.850	0.824	8.179
	800 K	0.822	0.807	0.823	0.809	0.816	0.799	0.781	0.808	0.816	0.801	8.082
	1.6 M	0.834	0.811	0.771	0.813	0.795	0.804	0.800	0.828	0.837	0.803	8.096
	3.2 M	0.857	0.805	0.812	0.792	0.778	0.819	0.824	0.836	0.835	0.809	8.167
	6.4 M	0.869	0.825	0.806	0.815	0.831	0.796	0.797	0.805	0.819	0.837	8.200
	12.8 M	0.939	0.824	0.795	0.811	0.816	0.802	0.819	0.841	0.825	0.809	8.281

1 Column with dictionary of size 12.8M - Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	27.260	16.140	16.163	16.140	15.934	16.107	16.619	16.023	15.856	15.842	172.084
	200 K	24.312	14.922	15.243	14.965	15.012	14.887	14.885	18.105	16.829	14.915	164.075
	400 K	22.995	14.115	14.127	14.147	14.189	14.333	14.084	14.894	14.137	14.702	151.723
	800 K	20.776	12.858	12.864	13.010	13.125	12.813	12.808	13.005	12.844	13.295	137.398
	1.6 M	19.933	11.622	11.915	11.721	11.651	11.551	11.574	11.733	11.573	11.621	124.894
	3.2 M	18.908	11.219	11.665	11.270	11.303	11.523	11.182	11.361	11.186	11.216	120.833
	6.4 M	18.269	11.558	11.905	11.625	11.635	11.512	11.547	11.688	11.492	11.526	122.757
	12.8 M	17.190	11.732	11.810	11.982	11.980	11.696	11.714	11.874	11.726	11.740	123.444

1 Column with dictionary of size 12.8M - Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	28.084	16.944	16.996	16.945	16.756	16.929	17.405	16.843	16.645	16.630	180.177
	200 K	25.122	15.703	16.039	15.761	15.821	15.674	15.693	18.914	17.633	15.723	172.083
	400 K	23.803	14.927	14.918	14.943	14.984	15.147	14.926	15.741	14.987	15.526	159.902
	800 K	21.598	13.665	13.687	13.819	13.941	13.612	13.589	13.813	13.660	14.096	145.480
	1.6 M	20.767	12.433	12.686	12.534	12.446	12.355	12.374	12.561	12.410	12.424	132.990
	3.2 M	19.765	12.024	12.477	12.062	12.081	12.342	12.006	12.197	12.021	12.025	129.000
	6.4 M	19.138	12.383	12.711	12.440	12.466	12.308	12.344	12.493	12.311	12.363	130.957
	12.8 M	18.129	12.556	12.605	12.793	12.796	12.498	12.533	12.715	12.551	12.549	131.725

Map Buffer

1 Column with dictionary of size 12.8M - Insertions												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	0.559	0.573	0.557	0.555	0.537	0.554	0.565	0.530	0.575	0.566	5.571
	200 K	0.550	0.573	0.613	0.547	0.598	0.549	0.583	0.597	0.582	0.552	5.744
	400 K	0.586	0.525	0.594	0.543	0.592	0.563	0.569	0.550	0.574	0.570	5.666
	800 K	0.561	0.568	0.571	0.559	0.557	0.574	0.592	0.612	0.574	0.567	5.735
	1.6 M	0.570	0.589	0.550	0.595	0.568	0.597	0.586	0.583	0.574	0.579	5.791
	3.2 M	0.576	0.567	0.551	0.579	0.584	0.590	0.560	0.579	0.568	0.583	5.737
	6.4 M	0.614	0.592	0.555	0.550	0.565	0.554	0.568	0.570	0.555	0.578	5.701
	12.8 M	0.687	0.586	0.561	0.527	0.569	0.588	0.548	0.543	0.537	0.604	5.750

1 Column with dictionary of size 12.8M - Merge												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	53.577	37.493	36.967	36.385	37.629	36.118	36.308	37.497	42.824	40.605	395.403
	200 K	57.927	40.136	40.082	38.935	38.796	38.498	38.997	38.472	38.539	38.696	409.078
	400 K	57.525	41.277	43.657	41.609	41.738	42.254	42.203	42.150	40.964	41.069	434.446
	800 K	59.979	44.704	45.131	43.341	43.775	43.285	42.992	43.972	43.262	43.926	454.367
	1.6 M	62.395	46.742	45.317	46.751	46.469	47.351	45.481	45.369	48.106	45.599	479.580
	3.2 M	65.782	49.586	48.525	47.735	49.946	49.362	47.581	48.228	48.105	48.719	503.569
	6.4 M	67.070	51.619	50.488	51.170	50.474	50.544	50.378	50.638	50.386	54.737	527.504
	12.8 M	70.515	53.889	55.368	54.291	56.952	53.867	54.262	57.846	54.343	56.534	567.867

1 Column with dictionary of size 12.8M - Total												
		Size Delta 2										Total Time
		12.8 M	25.6 M	38.4 M	51.2 M	64 M	76.8 M	89.6 M	102.4 M	115.2 M	128 M	
Size Delta 1	100 K	54.136	38.066	37.524	36.940	38.166	36.672	36.873	38.027	43.399	41.171	400.974
	200 K	58.477	40.709	40.695	39.482	39.394	39.047	39.580	39.069	39.121	39.248	414.822
	400 K	58.111	41.802	44.251	42.152	42.330	42.817	42.772	42.700	41.538	41.639	440.112
	800 K	60.540	45.272	45.702	43.900	44.332	43.859	43.584	44.584	43.836	44.493	460.102
	1.6 M	62.965	47.331	45.867	47.346	47.037	47.948	46.067	45.952	48.680	46.178	485.371
	3.2 M	66.358	50.153	49.076	48.314	50.530	49.952	48.141	48.807	48.673	49.302	509.306
	6.4 M	67.684	52.211	51.043	51.720	51.039	51.098	50.946	51.208	50.941	55.315	533.205
	12.8 M	71.202	54.475	55.929	54.818	57.521	54.455	54.810	58.389	54.880	57.138	573.617

11.2. Query results

25,000 values in dictionary

Select

1 Column with 25 K different values									
	Rows inserted								Total Time
	100 K	200 K	400 K	800 K	1.6 M	3.2 M	6.4 M	12.8 M	
Delta 2	0.008	0.015	0.029	0.058	0.116	0.257	0.504	0.999	1.986
Index Buffer	0.014	0.027	0.054	0.109	0.222	0.461	0.927	1.901	3.715
Standard Buffer	0.092	0.186	0.387	0.847	1.691	3.075	6.543	13.208	26.029
Sort Buffer	0.091	0.197	0.394	0.785	1.571	3.139	6.991	12.918	26.086

Scan

1 Column with 25 K different values									
	Rows inserted								Total Time
	100 K	200 K	400 K	800 K	1.6 M	3.2 M	6.4 M	12.8 M	
Delta 2	0.935	2.031	4.074	8.432	17.437	34.357	72.040	155.865	295.171
Index Buffer	1.518	2.889	5.136	10.090	20.705	37.881	79.549	169.514	327.282
Standard Buffer	1.556	2.826	5.510	10.676	20.358	38.363	79.094	171.545	329.928
Sort Buffer	1.588	2.995	5.460	10.700	21.927	41.087	79.515	172.941	336.213

Range

1 Column with 25 K different values									
	Rows inserted								Total Time
	100 K	200 K	400 K	800 K	1.6 M	3.2 M	6.4 M	12.8 M	
Delta 2	1.346	2.728	5.613	12.147	24.177	46.710	92.495	206.580	391.796
Index Buffer	1.653	3.279	6.688	12.285	26.821	50.935	99.358	219.241	420.260
Standard Buffer	1.541	3.311	5.778	11.066	21.619	44.366	78.942	181.739	348.362
Sort Buffer	1.657	3.192	6.346	12.854	22.807	45.431	98.469	174.170	364.926

50,000 values in dictionary

Select

1 Column with 50 K different values									
	Rows inserted								Total Time
	100 K	200 K	400 K	800 K	1.6 M	3.2 M	6.4 M	12.8 M	
Delta 2	0.008	0.015	0.030	0.058	0.115	0.252	0.495	0.985	1.958
Index Buffer	0.014	0.042	0.055	0.109	0.218	0.459	0.918	1.824	3.639
Standard Buffer	0.090	0.185	0.386	0.772	1.538	3.073	6.370	13.219	25.633
Sort Buffer	0.091	0.197	0.394	0.786	1.570	3.136	6.986	13.690	26.850

Scan

1 Column with 50 K different values									
	Rows inserted								Total Time
	100 K	200 K	400 K	800 K	1.6 M	3.2 M	6.4 M	12.8 M	
Delta 2	0.937	1.975	4.082	8.382	17.349	34.790	73.241	161.035	301.791
Index Buffer	1.497	2.710	5.204	10.155	19.884	39.011	78.167	169.481	326.109
Standard Buffer	1.539	2.841	5.289	10.521	20.052	38.228	78.354	171.457	328.281
Sort Buffer	1.823	2.988	6.752	10.711	21.668	41.389	79.731	180.489	345.551

Range

1 Column with 50 K different values									
	Rows inserted								Total Time
	100 K	200 K	400 K	800 K	1.6 M	3.2 M	6.4 M	12.8 M	
Delta 2	1.587	2.928	6.178	12.233	24.253	53.245	96.404	193.682	390.510
Index Buffer	2.026	3.735	6.099	13.177	24.994	47.134	107.802	208.618	413.585
Standard Buffer	1.685	3.190	5.700	11.018	20.659	44.535	88.043	177.150	351.980
Sort Buffer	1.627	3.506	7.318	12.635	24.465	50.435	92.323	209.880	402.189

100,000 values in dictionary

Select

1 Column with 100 K different values									
	Rows inserted								Total Time
	100 K	200 K	400 K	800 K	1.6 M	3.2 M	6.4 M	12.8 M	
Delta 2	0.007	0.014	0.029	0.057	0.114	0.250	0.492	0.978	1.941
Index Buffer	0.013	0.027	0.055	0.110	0.217	0.458	0.956	1.825	3.661
Standard Buffer	0.090	0.186	0.387	0.768	1.536	3.254	6.107	12.408	24.736
Sort Buffer	0.090	0.196	0.392	0.782	1.568	3.136	6.858	13.361	26.383

Scan

1 Column with 100 K different values									
	Rows inserted								Total Time
	100 K	200 K	400 K	800 K	1.6 M	3.2 M	6.4 M	12.8 M	
Delta 2	0.954	1.997	4.220	8.548	17.507	34.787	72.828	167.267	308.108
Index Buffer	1.468	2.708	5.538	10.045	20.290	38.257	79.072	169.530	326.908
Standard Buffer	1.544	2.835	5.278	10.411	20.199	38.993	78.660	170.477	328.397
Sort Buffer	1.651	3.092	5.464	10.992	21.749	41.530	79.698	179.912	344.088

Range

1 Column with 100 K different values									
	Rows inserted								Total Time
	100 K	200 K	400 K	800 K	1.6 M	3.2 M	6.4 M	12.8 M	
Delta 2	1.795	3.351	6.560	12.096	23.808	44.690	104.870	211.228	408.398
Index Buffer	2.196	3.840	6.370	14.677	27.791	56.234	107.928	193.397	412.433
Standard Buffer	1.600	3.430	5.643	11.223	22.287	45.639	88.824	181.115	359.761
Sort Buffer	1.522	3.437	6.404	12.150	25.323	45.570	90.895	190.237	375.538

200,000 values in dictionary

Select

1 Column with 200 K different values									
	Rows inserted								Total Time
	100 K	200 K	400 K	800 K	1.6 M	3.2 M	6.4 M	12.8 M	
Delta 2	0.007	0.014	0.029	0.055	0.114	0.293	0.489	0.972	1.973
Index Buffer	0.013	0.028	0.055	0.109	0.216	0.456	0.910	1.821	3.608
Standard Buffer	0.090	0.186	0.385	0.768	1.536	3.066	6.086	12.439	24.556
Sort Buffer	0.090	0.196	0.393	0.786	1.566	3.129	6.559	12.675	25.394

Scan

1 Column with 200 K different values									
	Rows inserted								Total Time
	100 K	200 K	400 K	800 K	1.6 M	3.2 M	6.4 M	12.8 M	
Delta 2	0.972	1.979	4.074	8.454	17.981	35.105	73.171	165.088	306.824
Index Buffer	1.429	2.837	5.351	9.969	19.823	38.456	77.599	171.746	327.210
Standard Buffer	1.614	2.839	5.287	10.270	20.091	38.148	78.711	171.442	328.402
Sort Buffer	1.564	2.992	5.542	10.814	21.715	41.428	79.235	181.420	344.710

Range

1 Column with 200 K different values									
	Rows inserted								Total Time
	100 K	200 K	400 K	800 K	1.6 M	3.2 M	6.4 M	12.8 M	
Delta 2	1.734	3.372	6.721	13.179	25.674	48.834	104.249	218.836	422.599
Index Buffer	1.880	3.950	7.207	15.022	27.886	55.118	97.448	240.500	449.011
Standard Buffer	1.665	3.322	5.784	11.503	22.311	44.542	78.396	181.171	348.694
Sort Buffer	1.852	3.355	6.521	11.941	24.695	46.547	78.309	175.843	349.063

400,000 values in dictionary

Select

1 Column with 400 K different values									
	Rows inserted								Total Time
	100 K	200 K	400 K	800 K	1.6 M	3.2 M	6.4 M	12.8 M	
Delta 2	0.008	0.015	0.028	0.057	0.113	0.249	0.489	0.971	1.930
Index Buffer	0.014	0.028	0.055	0.108	0.219	0.456	0.907	1.818	3.605
Standard Buffer	0.089	0.186	0.385	0.767	1.533	3.042	6.082	13.140	25.224
Sort Buffer	0.090	0.196	0.392	0.783	1.566	3.410	6.618	12.837	25.892

Scan

1 Column with 400 K different values									
	Rows inserted								Total Time
	100 K	200 K	400 K	800 K	1.6 M	3.2 M	6.4 M	12.8 M	
Delta 2	0.946	1.981	4.029	8.513	17.921	34.668	73.216	166.716	307.990
Index Buffer	1.475	2.736	5.183	9.921	19.896	37.849	77.138	167.929	322.127
Standard Buffer	1.547	2.973	5.272	10.352	20.109	38.088	78.862	171.993	329.196
Sort Buffer	1.636	2.977	5.548	10.841	21.743	41.279	79.689	180.573	344.286

Range

1 Column with 400 K different values									
	Rows inserted								Total Time
	100 K	200 K	400 K	800 K	1.6 M	3.2 M	6.4 M	12.8 M	
Delta 2	1.724	3.659	8.044	14.886	29.011	53.433	108.089	212.349	431.195
Index Buffer	2.113	4.074	8.290	15.009	29.121	57.057	103.377	225.316	444.357
Standard Buffer	1.387	3.044	5.811	12.444	22.329	47.012	78.563	181.377	351.967
Sort Buffer	1.556	3.262	6.991	12.294	23.579	43.248	83.912	182.247	357.089

800,000 values in dictionary

Select

1 Column with 800 K different values									
	Rows inserted								Total Time
	100 K	200 K	400 K	800 K	1.6 M	3.2 M	6.4 M	12.8 M	
Delta 2	0.008	0.014	0.029	0.057	0.113	0.274	0.487	0.971	1.953
Index Buffer	0.014	0.027	0.055	0.108	0.217	0.475	0.931	1.815	3.642
Standard Buffer	0.090	0.186	0.385	0.766	1.532	3.055	6.106	13.183	25.303
Sort Buffer	0.091	0.195	0.392	0.890	1.564	3.215	6.507	12.634	25.488

Scan

1 Column with 800 K different values									
	Rows inserted								Total Time
	100 K	200 K	400 K	800 K	1.6 M	3.2 M	6.4 M	12.8 M	
Delta 2	0.951	2.068	4.144	8.423	17.450	35.379	73.745	165.561	307.721
Index Buffer	1.446	2.675	4.876	9.889	19.847	38.073	77.252	168.835	322.893
Standard Buffer	1.542	2.838	5.256	10.513	20.206	38.665	79.354	171.363	329.737
Sort Buffer	1.613	2.978	5.562	11.085	21.989	41.252	79.472	182.188	346.139

Range

1 Column with 800 K different values									
	Rows inserted								Total Time
	100 K	200 K	400 K	800 K	1.6 M	3.2 M	6.4 M	12.8 M	
Delta 2	1.855	3.388	8.035	14.401	27.722	57.774	108.651	217.817	439.643
Index Buffer	2.050	4.523	7.649	17.523	30.298	59.838	106.819	240.647	469.347
Standard Buffer	1.400	3.081	6.015	11.726	23.413	41.852	89.822	174.569	351.878
Sort Buffer	1.584	3.520	6.570	12.316	22.991	45.519	77.402	185.044	354.946

1,600,000 values in dictionary

Select

1 Column with 1.6 M different values									
	Rows inserted								Total Time
	100 K	200 K	400 K	800 K	1.6 M	3.2 M	6.4 M	12.8 M	
Delta 2	0.007	0.014	0.028	0.057	0.113	0.247	0.486	0.972	1.924
Index Buffer	0.013	0.028	0.054	0.108	0.222	0.454	0.903	1.819	3.601
Standard Buffer	0.090	0.186	0.385	0.766	1.532	3.046	6.132	12.739	24.876
Sort Buffer	0.091	0.195	0.392	0.782	1.564	3.444	6.287	12.668	25.423

Scan

1 Column with 1.6 M different values									
	Rows inserted								Total Time
	100 K	200 K	400 K	800 K	1.6 M	3.2 M	6.4 M	12.8 M	
Delta 2	0.952	1.982	4.061	8.884	17.562	34.694	72.749	165.566	306.450
Index Buffer	1.503	3.082	4.875	9.702	19.368	37.458	78.140	168.653	322.781
Standard Buffer	1.544	2.833	5.295	10.335	20.615	38.658	79.067	172.091	330.438
Sort Buffer	1.750	3.058	5.567	10.889	21.737	41.009	79.680	181.382	345.072

Range

1 Column with 1.6 M different values									
	Rows inserted								Total Time
	100 K	200 K	400 K	800 K	1.6 M	3.2 M	6.4 M	12.8 M	
Delta 2	1.673	3.694	7.686	15.393	31.461	54.127	117.233	221.921	453.188
Index Buffer	2.241	4.945	8.225	16.637	31.687	55.112	114.778	237.864	471.489
Standard Buffer	1.390	3.127	5.770	11.284	21.962	43.562	85.767	196.500	369.362
Sort Buffer	1.593	3.330	6.302	12.480	24.034	42.909	84.667	208.414	383.729

3,200,000 values in dictionary

Select

1 Column with 3.2 M different values									
	Rows inserted								Total Time
	100 K	200 K	400 K	800 K	1.6 M	3.2 M	6.4 M	12.8 M	
Delta 2	0.007	0.014	0.028	0.057	0.113	0.247	0.486	0.971	1.923
Index Buffer	0.013	0.028	0.054	0.108	0.223	0.453	0.922	1.807	3.608
Standard Buffer	0.090	0.185	0.384	0.766	1.665	3.055	6.108	13.273	25.526
Sort Buffer	0.090	0.196	0.393	0.783	1.562	3.585	6.611	12.642	25.862

Scan

1 Column with 3.2 M different values									
	Rows inserted								Total Time
	100 K	200 K	400 K	800 K	1.6 M	3.2 M	6.4 M	12.8 M	
Delta 2	0.963	1.991	4.103	8.404	17.421	34.433	73.710	165.591	306.616
Index Buffer	1.476	2.640	4.876	9.831	19.715	37.532	76.615	167.224	319.909
Standard Buffer	1.615	2.908	5.281	10.253	20.461	38.807	78.750	171.313	329.388
Sort Buffer	1.670	2.975	5.544	10.840	21.668	41.061	78.709	180.528	342.995

Range

1 Column with 3.2 M different values									
	Rows inserted								Total Time
	100 K	200 K	400 K	800 K	1.6 M	3.2 M	6.4 M	12.8 M	
Delta 2	1.697	3.592	6.870	14.446	29.913	61.824	111.284	221.757	451.383
Index Buffer	2.109	3.744	8.885	15.658	35.636	65.909	122.001	220.841	474.783
Standard Buffer	1.496	3.252	5.786	11.496	21.342	44.966	99.902	200.463	388.703
Sort Buffer	1.621	3.424	6.524	12.950	26.505	46.499	92.200	191.167	380.890

6,400,000 values in dictionary

Select

1 Column with 6.4 M different values									
	Rows inserted								Total Time
	100 K	200 K	400 K	800 K	1.6 M	3.2 M	6.4 M	12.8 M	
Delta 2	0.007	0.014	0.028	0.057	0.114	0.247	0.484	0.966	1.917
Index Buffer	0.014	0.027	0.055	0.109	0.219	0.453	0.908	1.815	3.600
Standard Buffer	0.090	0.185	0.385	0.767	1.531	3.057	6.102	13.172	25.289
Sort Buffer	0.093	0.202	0.404	0.814	1.614	3.225	6.449	12.893	25.694

Scan

1 Column with 6.4 M different values									
	Rows inserted								Total Time
	100 K	200 K	400 K	800 K	1.6 M	3.2 M	6.4 M	12.8 M	
Delta 2	0.999	1.977	4.199	8.469	17.306	34.484	73.066	165.776	306.276
Index Buffer	1.517	2.753	4.869	9.713	19.434	37.125	76.601	170.503	322.515
Standard Buffer	1.546	2.910	5.288	10.320	20.057	38.353	78.806	171.982	329.262
Sort Buffer	1.713	3.021	5.545	11.066	21.576	41.334	79.063	172.021	335.339

Range

1 Column with 6.4 M different values									
	Rows inserted								Total Time
	100 K	200 K	400 K	800 K	1.6 M	3.2 M	6.4 M	12.8 M	
Delta 2	1.625	3.767	7.591	14.935	31.915	69.812	128.956	254.937	513.538
Index Buffer	2.191	4.140	8.067	17.972	32.156	63.494	130.736	247.026	505.782
Standard Buffer	1.430	2.720	6.313	11.660	21.557	43.361	79.762	174.840	341.643
Sort Buffer	1.713	3.653	6.273	12.656	24.046	51.544	87.637	175.673	363.195

12,800,000 values in dictionary

Select

1 Column with 12.8 M different values									
	Rows inserted								Total Time
	100 K	200 K	400 K	800 K	1.6 M	3.2 M	6.4 M	12.8 M	
Delta 2	0.007	0.015	0.029	0.057	0.113	0.247	0.675	0.969	2.112
Index Buffer	0.014	0.027	0.054	0.108	0.220	0.453	0.904	1.813	3.593
Standard Buffer	0.090	0.196	0.384	0.766	1.532	3.061	6.231	12.576	24.836
Sort Buffer	0.097	0.215	0.405	0.813	1.614	3.221	6.426	12.873	25.664

Scan

1 Column with 12.8 M different values									
	Rows inserted								Total Time
	100 K	200 K	400 K	800 K	1.6 M	3.2 M	6.4 M	12.8 M	
Delta 2	0.971	1.990	4.053	8.662	17.368	34.722	73.094	164.872	305.732
Index Buffer	1.508	2.638	4.884	9.781	19.570	37.619	76.165	166.423	318.588
Standard Buffer	1.768	3.155	5.280	10.329	20.091	38.299	79.138	171.845	329.905
Sort Buffer	1.811	3.150	5.646	11.107	21.581	41.403	80.749	172.813	338.260

Range

1 Column with 12.8 M different values									
	Rows inserted								Total Time
	100 K	200 K	400 K	800 K	1.6 M	3.2 M	6.4 M	12.8 M	
Delta 2	1.703	3.514	8.255	15.918	30.687	68.926	134.782	299.919	563.704
Index Buffer	2.210	3.501	8.641	17.701	35.671	63.528	140.194	299.151	570.597
Standard Buffer	1.372	2.767	6.205	11.377	21.205	42.591	86.823	175.242	347.582
Sort Buffer	1.584	3.552	6.540	12.751	23.796	45.075	88.643	178.031	359.972